AFRL-RI-RS-TR-2010-225

PRIVATE INFORMATION RETRIEVAL

UNIVERSITY OF WISCONSIN

December 2010

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■**UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

FOR THE DIRECTOR:

/s/
FRANK BORN
Work Unit Manager

/s/
WARREN H. DEBANY JR., Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

1. REPORT DATE *(DD-MM-YYYY)* December 2010	**2. REPORT TYPE** Final Technical Report	**3. DATES COVERED** *(From - To)* January 2009 – July 2010

4. TITLE AND SUBTITLE PRIVATE INFORMATION RETRIEVAL	**5a. CONTRACT NUMBER** FA8750-09-2-0066
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Somesh Jha, Vitaly Shmatikov, Matthew Fredrikson	**5d. PROJECT NUMBER** NICE
	5e. TASK NUMBER 00
	5f. WORK UNIT NUMBER 18
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Wisconsin, System Research & Sponsored Programs, 21 N. Park Street, STE 6401, Madison WI 53715-1218	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory, RIGA, 525 Brooks Road, Rome NY 13441-4505; Office of the Director of National Intelligence, Intelligence Advanced Research Projects Agency, Washington DC 20511	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TR-2010-225

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report describes the IARPA sponsored private information retrieval (PIR) project. The approach is based on the keyword-oblivious transfer cryptographic primitive, which allows a client and server to negotiate an exchange of data based on a keyword not learned by the server. Although no protocols exist that allow this primitive to scale to the magnitude needed by PIR, we utilize a semi-trusted third party to meet the stated requirements. We evaluated this approach against a 60 gigabyte database and a set of queries provided by the MIT-LL test team. This approach exceeded the given performance requirements, with most of the performance penalty coming from encryption and decryption, rather than keyword-oblivious transfer.

15. SUBJECT TERMS
Private Information Retrieval, Privacy, MySQL, Private Search, PRI

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** FRANK BORN
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	40	**19b. TELEPHONE NUMBER** (***Include area code***) N/A

Table of Contents

List of Figures

1 Executive Summary

In this report, we describe our activities related to the private information retrieval (PIR) project. Our approach is based on the efficient implementations of the keyword-oblivious transfer cryptographic primitive, which allows a client and server to negotiate an exchange of data based on a keyword not learned by the server. Although no protocols exist that allow this primitive to scale to the magnitude needed by PIR, we utilize a semi-trusted third party, known as the isolated box, to meet the stated requirements. We implemented our approach in a realistic prototype, and evaluated its performance over a large (60 gigabyte) database and a set of queries provided by the MIT-LL test team. We found that our approach meets and exceeds the given performance requirements, with the majority of the performance penalty over plain MySQL processing coming from encryption and decryption, rather than keyword-oblivious transfer.

2 Introduction

The goals of the Automatic Privacy Protection program are to "*develop and demonstrate practical, sound automated methods for the use of private information retrieval techniques in Intelligence Community systems, to automatically protect the private data of untargeted individuals, to assure the mandated policies are enforced, and to enable more effective interagency and intergovernmental data sharing for improved security*." To this end, we have pursued the development of efficient protocols for the Private Information Retrieval (PIR) problem: a client queries a large-scale database on a potentially adversarial server, and learns the correct answer to his query without leaking any information about it to the server. We have demonstrated formally that our PIR protocol meets the stated privacy needs of IARPA, and produced a working prototype. A team of independent testers from MIT-Lincoln Labs has verified that our prototype is functional and bug-free on a large test corpus, and that is exceeds IARPA's minimum performance requirements by more than an order of magnitude. Our prototype is even more efficient since the MIT-LL test was conducted.

3 Technical Approach

As per the rules of engagement (ROE), our system has three primary components: a client, a server, and an isolated box.

- The server holds a plaintext database, consisting of an arbitrary number of rows organized according to a single schema. It communicates with the client and isolated box to provide responses to queries in an oblivious fashion.
  Assumptions: It is assumed that the server is honest-but-curious; it follows the protocol, and does not collude with the isolated box, but may attempt to learn more about the contents of the client's queries by running additional algorithms over its view (messages exchanged during the protocol).
  Guarantees: The server learns no information from processing a query. The keyword oblivious transfer (KOT) protocol used to match a query guarantees that the server does

not learn either the field over which the query is performed, or the field value targeted by the query. Furthermore, after completing the protocol, the server has provided the client with the information needed to retrieve, in clear text, exactly the rows of the database corresponding to the query.

• The client issues queries to the server, which take the form of a single attribute-value pair for each query. The attribute is an element of the schema, and the value is to be matched by each row in the query's result set.
Assumptions: It is assumed that the client is honest-but-curious, and does not adaptively select queries to learn more about the database than intended by the policy.
Guarantees: Upon completing a query, the client learns the following information about the database: the number of rows matching its query, the plaintext of each matching row, and the size of the database. It does not learn the plaintext of any rows not matching its query, and there are no rows in the database that match its query for which it does not learn the plaintext.

• The isolated box maintains an encrypted, permuted version of the server's database. Intuitively, the isolated box serves as an oblivious storage point that allows our protocol to optimize the amount of network traffic transferred in the course of serving a query.
Assumptions: It is assumed that the isolated box is honest-but-curious. It does not collude with the client to learn more about the server's database, and it does not collude with the server to learn more about the client's query.
Guarantees: The isolated box can learn the approximate frequency with which an individual encrypted, permuted record of the database is accessed. Note that the isolated box does *not* learn the contents of the record and the frequency-of-access information is not perfect due to randomness introduced by the client.

3.1 Definitions

A database D is a set of records indexed by t attributes $A_1, \ldots, A_t$, where we identify an attribute A_i with the set of attribute values it may take. Each record r of the database takes the form $r = (x_1, \ldots, x_t, y)$ with each x_i in A denoting an attribute value, and y in $\{0, 1\}^l$ (for some length parameter l) being the payload. We assume a database contains no duplicate records. Queries are of the form (i, x) where $1 \leq i \leq t$ and x in A_i. The query $q = (i, x)$ represents a request for all records whose value in the ith attribute is x. (For such a query, we call i the *relevant attribute* and x the *keyword*.) Formally, for the query $q = (i, x)$ on a database D we define $q(D) = \{(x_1, \ldots, x_t, y) \text{ in } D \mid x_i = x\}$ as the set of records that match the query.

3.2 Primitives

We assume a semantically-secure encryption scheme $E = (E, U)$ defined over (K, M, C), where K is the keyspace, M is the space of plaintexts, and C is the space of ciphertexts. Additionally, we assume a cryptographically-secure hash function, and previously-established RSA credentials (N, e, d) for the server (N is the modulus, e is the public exponent, and d is the private key). Finally, we assume a keyword oblivious transfer (KOT) scheme $OT^{k_1,\dots,k_m}$, the security of which is based on the intractability of the one-more-RSA-inversion problem.

3.3 PIR Protocol

The protocol works in three stages. In the preprocessing stage, S initializes IB with information from D. In the second (query) stage, C, S, and IB communicate to serve a query from C. The preprocessing stage is performed once before any queries are served, and periodically when needed to ensure the privacy of D according to refresh parameter r. The second is performed each time C has a new query. In the third (refresh) stage, a decision is made as to whether the preprocessing stage is re-executed to gain additional privacy for D.

• Preprocessing stage:

1. S does the following once, before any queries are served:
   - – Picks n random keys $k_1, \dots, k_n$ from K.
   - – Picks a random permutation s of n elements.
   - – Prepares n ciphertexts $c_1, \dots, c_n$, where $c_i = E_{k_i}(R_i)$.
   - – Sends the list $c\_1, \dots, c\_n$ to the isolated box IB.
   - – Initializes the refresh counter: cnt to 0.

• Query stage:

1. C and S perform a keyword oblivious transfer step. C's input is (i, x) (the entire query), and S's input is:

$$\{([i, x_{ij}], [K_{ij}, I_{ij}]) \mid 0 \le i < l, 0 \le j < |A_i|, x_{ij} \text{ in } A_i\} \cup P_D$$

where

$$K_{ij} = \{k_h \mid r_h = (x_{h,0}, \dots, x_{ij}, \dots, x_{h,l})\}$$

and

$$I_{ij} = \{s^{-1}(h) \mid r_h = (x_{h,0}, \dots, x_{ij}, \dots, x_{h,l})\}$$

and

$$P_D = \{([R_i, x_{R_i}], [k_{R_i}, I_{R_i}]) \mid 0 \leq i < l|D| - \sum_{j=0}^{l} |A_j| \wedge x_{R_i} \text{ not in } A_{R_i}\}$$

where R is a random sequence of integers between 0 and l, k is a random sequence of elements from K, and I is a random sequence of sets of database indices. Here the quantity $l|D| - \sum_{j=0}^{l} |A_j|$ refers to the number of additional cells that must be added to hide the distinct number of attribute-value pairs in the database. Note the assumption that K_{ij} , I_{ij} , k_{R_i} , and I_{R_i} are padded to the same length. The condition x_{R_i} not in A_{R_i} is to ensure that the entries added for padding will never be returned as the result of the KOT protocol. The purpose of including P_D in the server's input is to prevent leaking the number of distinct attribute-value pairs in the database. At the end of the KOT protocol, C receives $[K_{ij} , I_{ij}]$ such that $(i, x_{ij}) = (i, x)$. This is achieved using the keyword OT scheme.

2. C asks IB for the encrypted rows listed in I_{ij} .

3. IB sends C the ciphertexts indexed by I_{ij} , $\{c_{s(s^{-1}(h))} \mid s^{-1}(h) \text{ in } I_{ij}\}$

4. C can now use the elements of K_{ij} to decrypt those returned by IB, thus attaining:

$$\{(x_1, \ldots, x_t, y) \text{ in } D \mid x_i = x\}$$

5. S updates the refresh counter: $cnt \quad cnt + 1$.

• Refresh stage:

1. If $cnt > r$, then set $cnt \quad 0$ and re-execute the preprocessing stage.

3.4 Security Properties

It can be shown that our protocol does not reveal to the server: (1) the attribute over which the query is performed, (2) the value of the attribute queried for, or (3) the rows of the database which are accessed to the server. (1) and (2) follow directly from the guarantees of the KOT protocol that we use, and as such are subject to the same assumptions as that protocol (namely, intractability of the one-more-RSA-inversion problem). The third property follows from the fact that rows are only retrieved (during query processing) from the isolated box. Similarly for the client, this protocol does not leak any information aside from the intended result – the rows of the database that match the client's query. This property follows from the guarantees of the KOT

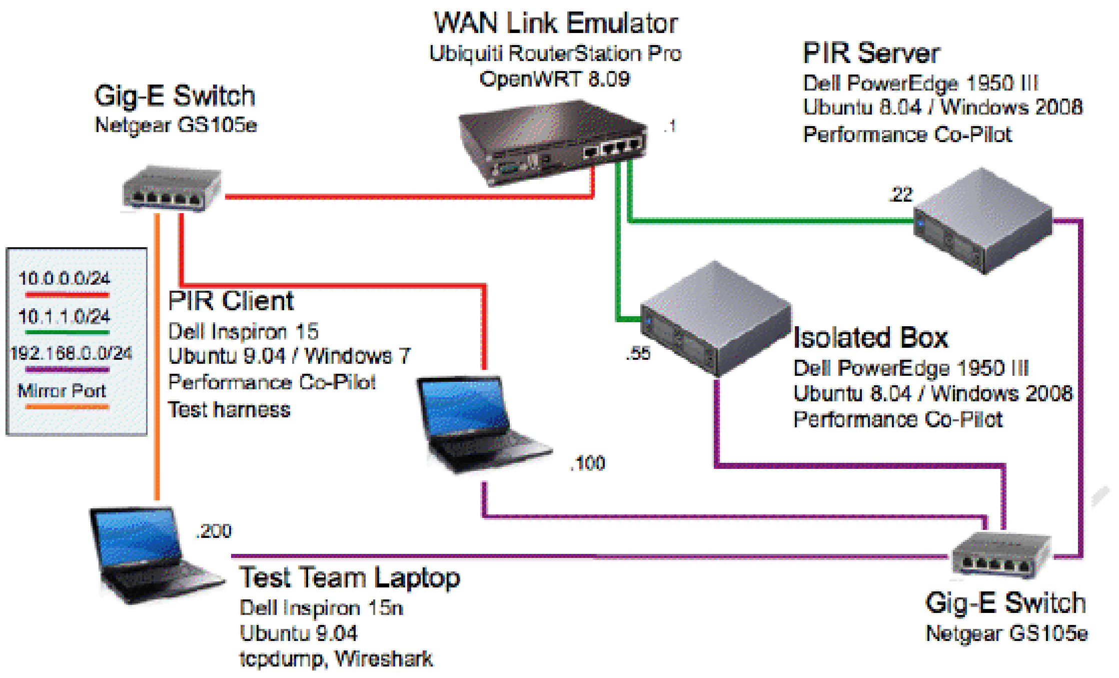

Figure 1: Experimental Setup. (Courtesy of MIT-LL)

protocol, as well as the security of the encryption scheme E. As such, privacy from an honest-but-curious client is subject to the same conditions on security as the KOT protocol. The privacy of D from the isolated box follows from the security of our encryption scheme E. However, the isolated box may learn the frequency with which the client accesses permuted rows of the database. Because the rows are permuted randomly before being sent to the isolated box, an adversary would need external (semantic) information in combination with this frequency information to deduce further information about the contents of the database. Furthermore, the adversarial utility of this information can be arbitrarily reduced by requiring the server to periodically re-send the rows to the isolated box using a fresh permutation. This feature is controlled by the parameter r; low values of r cause the server to enter the refresh phase more often. The more often this happens, the less useful the information learned by the isolated box becomes. However, each such refresh comes at the cost of substantial network overhead (for large databases), in addition to negative cache effects. This gives the protocol a tunable parameter: the refresh frequency offers various degrees of security for a quantifiable tradeoff in efficiency.

4 Lessons Learned

We feel that one noteworthy aspect of our work in private information retrieval is that we were able to scale to the requirements given by IARPA without developing any new cryptographic primitives (in fact, as we discuss in Section 2, the performance of our protocol outperformed the project goals by more than an order of magnitude). Recall that we rely on symmetric-key cryptography to efficiently hide data as it travels over un-trusted channels, keyword oblivious-transfer to hide the client's query from the server while retrieving the correct set of rows, and RSA to blind data within the keyword oblivious-transfer protocol. Particularly relevant is keyword oblivious-transfer: the high-level functionality of this primitive parallels that of private information retrieval so closely that implementing the needed functionality is merely a matter of scale. In other words, one could use keyword-oblivious transfer to implement private information retrieval, without further modification, were its performance at the scale mandated by IARPA acceptable. Our insight was to use keyword oblivious-transfer only over data that indexes relevant entries in the large database; when the client and server finish performing keyword oblivious-transfer, the client can use this information to ask the isolated box for the full information required to complete the private information retrieval protocol.

In one sense, this suggests that all of the mathematical tools needed to realize the demanding functionality of private information retrieval have existed for years. We see this as further evidence of the need for a new set of tools that *compile* privacy-sensitive programs from high-level specifications to low-level primitives with rigorously-proven properties, such as keyword-oblivious transfer. This will allow applications which have seemingly novel privacy requirements, such as private information retrieval, to benefit from principles developed in the software engineering community, such as code reuse, abstraction reuse, and low-level code generation. In the context of privacy-preserving applications, these principles have strong implications for correctness, as code/abstraction reuse oftentimes allow correctness proofs to be reused without loss of rigor. Removing the need to manually develop new correctness proofs for each protocol from the ground up is a major advantage. We see this as a primary advantage over other teams' solutions: re-using existing primitives to meet project requirements increased the clarity of our protocol description and correctness proof.

Our original proposal was based on the concept of an optimizing compiler for privacy-preserving applications. We view our activities with the private information retrieval protocol presented above as a case study in this larger effort. This project has provided us with a realistic application, corresponding evaluation dataset, and third-party testing. Moving forward, we will leverage this to incorporate the abstractions and functionality used to complete the project into such a compiler.

5 Implementation and Performance Evaluation

Implementation

We implemented our protocol in 10,501 lines of C++ source code for Linux. We use SQLite for back-end database processing, as it is lightweight, easy to use, and highly performant in the single-access, read-only setting. Our client prototype utilizes multiple threads to avoid network and encryption-related bottlenecks. One thread constantly transfers data from the isolated box over the network, and the other thread continually decrypts and displays the data. For most cryptographic primitives, we utilized the OpenSSL library, including 256-bit AES to store an encrypted copy of the database on the IB, and to generate secure pseudorandom numbers for key data and database row permutations. We wrote our own implementation of the Kurosawa-Ogata keyword oblivious transfer protocol, using 1024-bit RSA keys.

Experimental Setup

We evaluated the performance of our prototype experimentally. We loaded and ran the server and isolated box components onto two Dell PowerEdge servers, matching the project specification. The client was run on a Dell Inspiron 1545 matching the project specification. All communications took place over a local gigabit ethernet network. This setup is depicted in Figure 1.

Dataset and Benchmarks

The data that was used to perform the evaluation was provided by the MIT-Lincoln Labs test team. It consists of two components:

- A synthetic database with a schema corresponding to personnel records for a hypothetical company. The schema has 50 components arranged in a flat hierarchy, and 100,000 records corresponding to non-existent citizens with characteristics that fit the distribution found in 2000 census data. The total size of this database is approximately 60 gigabytes.
- 514 database queries arranged in 16 distinct test cases. These queries correspond to 182,348 database records, selected to test the full range of prototype operation.

Each test query consists of a SQL SELECT statement over a single attribute, with an equality constraint on the value of the attribute. For example,

```
SELECT * FROM people WHERE state = ‘NY’
```

To test different aspects of prototype performance, such as the ability to quickly begin returning data for a large query, the query attribute is varied to account for the characteristics of the underlying database. For example, querying sparse attributes allows the lookup performance of the prototype to be evaluated, without excess noise due to large result set transfer.

On average, test queries produce results with less than 10% of the records in the database. Test queries were broken into four categories:

- Tiny queries: fewer than 10 records.
- Small queries: between 10 and 1000 records.
- Medium queries: between 1000 and 10000 records.
- Large queries: greater than 10000 records.

The total benchmark suite contained 304 tiny queries, 224 small queries, 32 medium queries, and 4 large queries.

Metrics and Goals

The experiments tested four aspects of the implementation: correctness, query compilation time, index lookup and search time, and retrieval, decryption, and display time.

- **Correctness:** Because the requirements of PIR are stricter than those for traditional networked data retrieval, it is conceivable that the functionality of a PIR system might differ from a traditional system. For each test in the benchmark suite, we ran an identical test in a baseline MySQL installation to determine a baseline truth. We then checked the contents of each result against the baseline MySQL results, checking that both:

1. The PIR prototype returns the same number of records as the MySQL installation.
2. Each byte of each decrypted record returned by the PIR prototype matches the corresponding byte in the corresponding row returned by the MySQL installation.

- **Client Query Compilation (CQC) Time:** This corresponds to the period of time needed on the client to encode and send a query to the server.
- **Index Lookup and Search (ILS) Time:** This corresponds to the time needed for the client, server, and isolated box to negotiate the PIR protocol. This begins when the client's packet is first received by the server, and ends when the server's first result packet is sent.
- **Retrieval, Decryption, and Display (RDD) Time:** This corresponds to the time needed for the server to transfer all results to the client, as well as that needed by the client to decrypt and display the results. This period begins when the client outputs the first byte of the query result, and ends when all results have been displayed.

Each of these metrics is evaluated for each test query in the benchmark suite.

6 Results

Before we present the details of our results, we note that IARPA presented a number of performance requirements that the PIR prototype must meet.

1. The average index lookup and search time must be less than 60 seconds.
2. The average retrieval, decryption and display time of the PIR system must be no more than a factor of 100 more expensive than a corresponding baseline, non-PIR MySQL system.
3. The PIR system must take less than 24 hours to bring the entire 60 gigabyte test database online, ready to answer queries.

We are happy to report that our prototype meets and exceeds these requirements by substantial margins. To summarize, our results demonstrate that PIR can be made both practical and efficient. In particular:

- Bringing server and isolated box online is relatively inexpensive. For the 60 gigabyte dataset, it takes approximately three hours to bring all data online, and come to a ready state for query processing. There are two components to this cost: the transfer of permuted rows between the server and isolated box (~2.5 hours), and pre-computing the keyword oblivious transfer dictionary (~30 minutes).
- The overhead for performing keyword oblivious transfer is effectively constant, and nearly negligible. On average, for the full 60 gigabyte dataset, KOT required 4 seconds. This is significant, as performing KOT constitutes nearly all overhead required by PIR over standard query processing.
- Overall PIR query processing time is <2x the standard MySQL base time. For sufficiently large result sets, nearly all overhead is due to decryption time. This time can be reduced further with increased parallelism and faster encryption primitives.

- Our prototype returned correct results for all tests: both the number of records and the contents of each record matches that returned by the baseline MySQL installation.

A sampling of our results is displayed in Figure 2, which displays the query processing time for our PIR prototype versus the MySQL base time over tests from each category. The "unoptimized" curve corresponds to our prototype with no parallelism, and the "optimized" curve corresponds to the implementation with one additional processing thread.

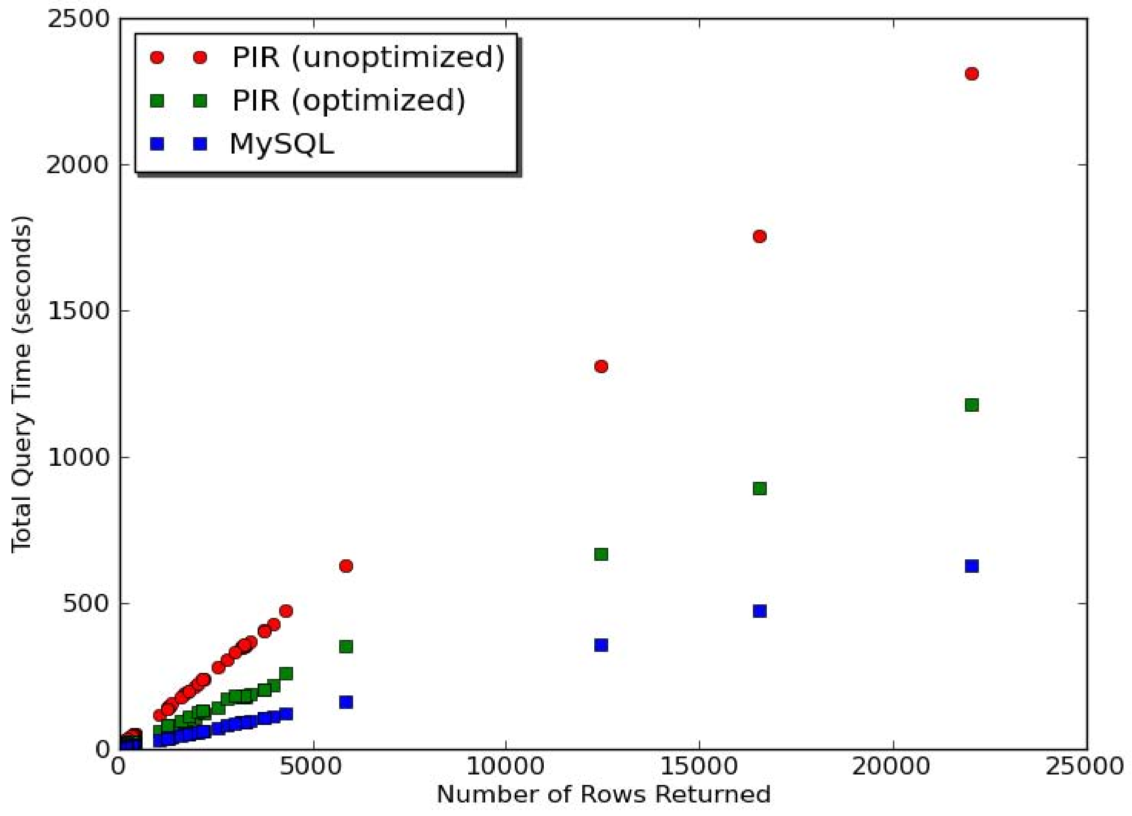

Figure 2: PIR Time vs. MySQL base time

We present more detailed results in the remainder of this section. In each plot, when a curve is fitted to the data, it is done so using least-squares regression. The coefficient of determination for each curve is labeled R^2. This coefficient takes values between 0 and 1; values near 1 indicate an excellent fit, and values near 0 indicate a weak correlation between the curve and data.

Tiny Queries
The tiny test set consists of 304 queries that return less than ten rows from the test database. Our results are given in four plots below. The first plot corresponds to the total query response time; note that the total query response time for our prototype is greater than that for the default MySQL installation. The second plot shows the client query compilation time. For tiny queries, the query compilation time of our prototype consumes nearly all of the total query response time. This reflects the fact that the time required to complete keyword oblivious-transfer is independent of the size of the query result. So, while the results to not take long to transmit from the isolated box to the server (reflected in the final plot), the query compilation time is as expensive as it would be for larger queries.

While it may seem as though this result implies that our protocol is not well-suited for tiny queries, observe that the total query response time is still small. We feel that the tradeoff is justified, given the scale of the data involved.

Note that the index lookup and search plot indicates no time needed by our prototype. This is due to the fact that client query compilation and index lookup or search correspond to the same portion of our protocol; the entire time is accounted for in the client query compilation data.

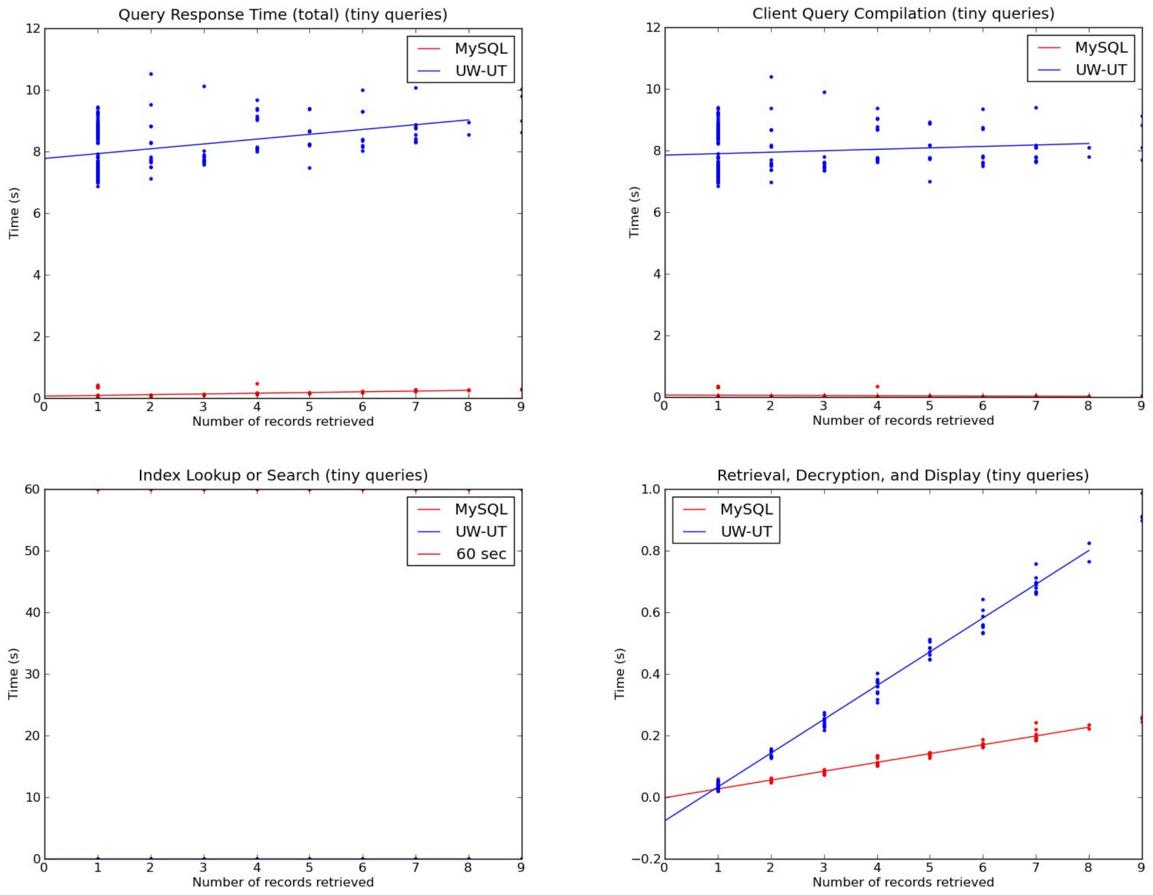

Figure 3: Response Time for Tiny Queries

Small Queries

The small test set consists of 224 queries that return between ten and 1000 records from the test database. Our results are given in four plots below. The first plot corresponds to the total query response time; note that the total query response time for our prototype is greater than that for the default MySQL installation, but only by a constant factor. This result is due to the need to decrypt results sent from the isolated box, and can be minimized to an arbitrary degree with increased parallelism and faster hardware. The second plot shows the client query compilation time. Note that the query compilation time for small queries falls in the same range as for tiny queries. This reflects the fact that keyword oblivious transfer time is independent of the query result size. Rather, it scales linearly in the number of records in the database, and the amount of value-repetition among the entries.

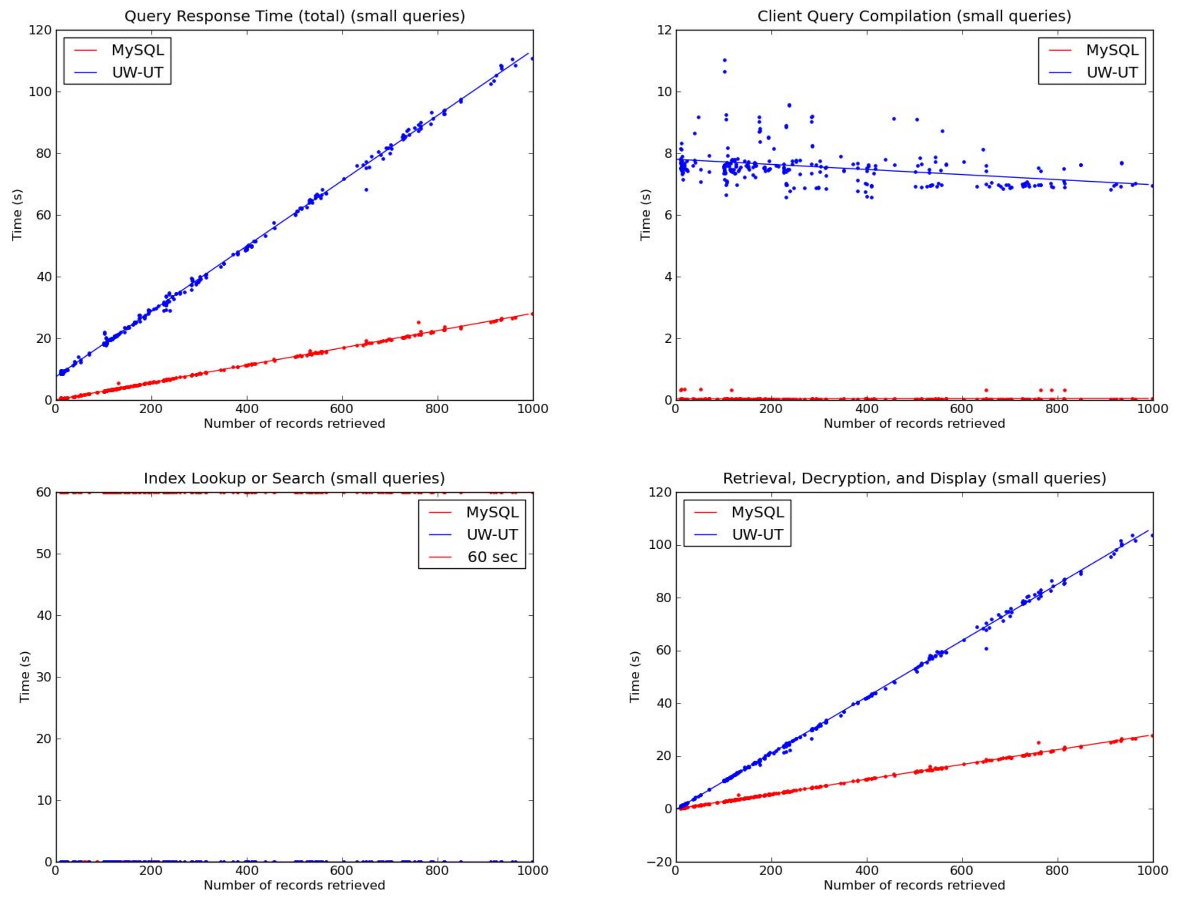

Figure 4: Response Time for Small Queries

Medium Queries

The medium test set consists of 32 queries that return between 1000 and 10 000 records from the test database. Our results are given in four plots below. The first plot corresponds to the total query response time. As in the previous set of tests cases, our prototype's total query response time differs from the baseline MySQL time by a constant factor. This effect can be minimized to an arbitrary degree using the same methods discussed previously. The second plot shows the client query compilation time. Note that the query compilation time for small queries falls in the same range as for tiny queries and small. This reflects the fact that keyword oblivious transfer time is independent of the query result size. However, for result sets of this size, query compilation corresponds to a much smaller portion of the overall response time, likewise mirrored in the striking similarities between the first and fourth plots below. At this point, the time needed to complete the keyword oblivious-transfer protocol is effectively marginalized by the time needed to transfer and decrypt the large amount of result data.

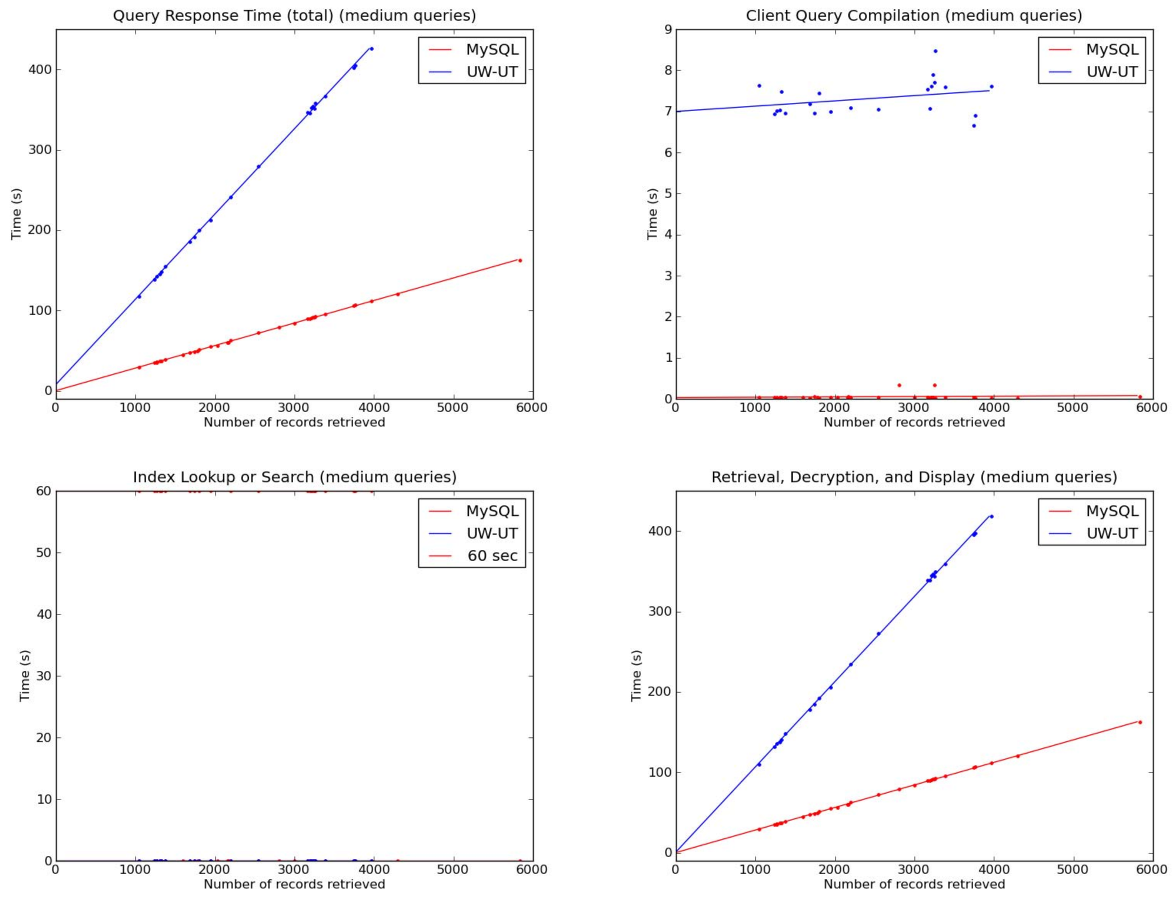

Figure 5: Response Time for Medium Queries

Large Queries

The large test set consists of 4 queries that return more than 10 000 records from the test database. Recall that the entire test database is 60 gigabytes, so each query in the large set corresponds to between five and ten gigabytes of response data. Our results are given in four plots below. The first plot corresponds to the total query response time. As in the previous set of tests cases, our prototype's total query response time differs from the baseline MySQL time by a constant factor. The second plot shows the client query compilation time. Note that the curve that fits these data points indicates a correlation between result size and query compilation time. This is almost surely a spurious effect of the small number of sample points; there is no reason to believe that query compilation time should differ at all from the other test sets. As with the medium tests, query compilation corresponds to a small portion of the overall response time.

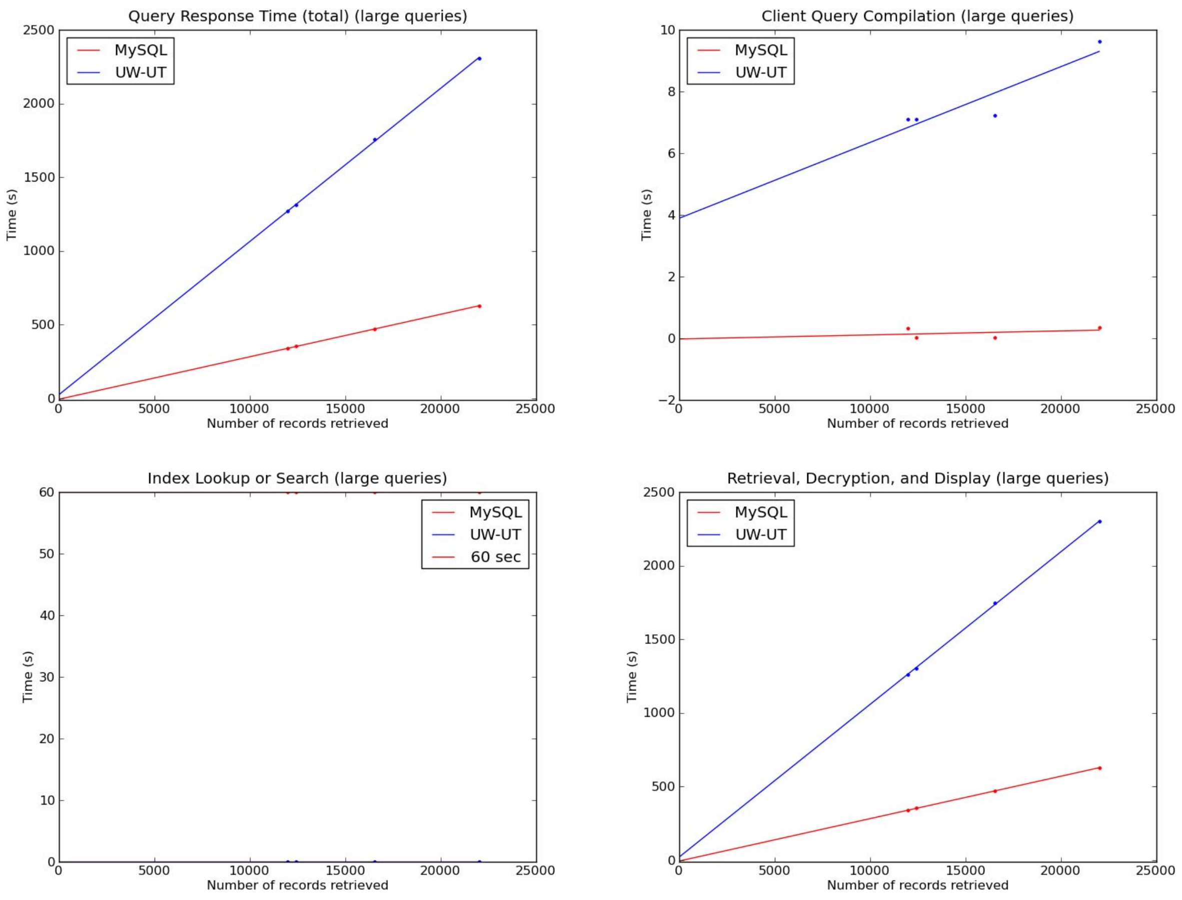

Figure 6: Response Time for Large Queries

7 Activities

We performed the following activities throughout the course of this project.

1. Developed initial protocol, which was based on pre-computing all possible database views for a particular attribute, and transferring them to the isolated box on-demand. This gives the isolated box a coarser view of database accesses, but is not nearly as performant as the approach discussed above.
2. Modified initial protocol to work over rows rather than views, to arrive at the approach discussed above.
3. Implemented the primitives needed by the protocol, including the Kurosawa-Ogata keyword-oblivious transfer protocol, the RSA algorithm, and the needed operations over large integers.
4. Implemented a prototype of the client, server, and isolated box in C++.
5. Integrated the prototype with the MIT-LL test harness.
6. Installed and configured the hardware infrastructure needed to run the performance evaluation.

8 Conclusion

We demonstrated an efficient protocol to perform keyword search in a privacy-preserving way. It was a very rewarding experience for our team. However, we want to pursue several future directions for this project. On the fundamentals side we want to explore support for more expressive queries. We want to see whether there are even more opportunities for making the protocol more efficient. University of Wisconsin has a very strong database group. We plan to collaborate with the database group to find more applications of the PIR technology. We are very excited about further opportunities on this project.

9 Acronyms

AES – Advanced Encryption System
CQC – Client Query Compilation
IARPA – Intelligence Advance Research Projects Agency
ILS – Index, Lookup and Search
KOT – Keyword Oblivious Transfer
MIT-LL – Massachusetts Institute of Technology Lincoln Labs
PIR – Private Information Retrieval
RSA - Rivest, Shamir and Adleman
SSL – Secure Socket Layer

10 References

S. Son, V. Shmatikov. "The Hitchhiker's Guide to DNS Cache Poisoning". In Proc. of 6th International ICST Conference on Security and Privacy in Communication Networks (SecureComm), 2010.

M. Bond, V. Srivastava, K. McKinley, V. Shmatikov. "Efficient, Context-Sensitive Detection of Real-World Semantic Attacks". Proc. of 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS), 2010.

I. Roy, S. Setty, A. Kilzer, V. Shmatikov, E. Witchel. "Airavat: Security and Privacy for MapReduce". Proc. of 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2010.

L. Kruger, M. Fredrikson, S. Jha, V. Shmatikov. “SSH Password Authentication Using Secure Function Evaluation”. Under Submission.

Matthew Fredrikson, Benjamin Livshits: RePriv: Re-Envisioning In-Browser Privacy. Microsoft Research Technical Report MSR-TR-2010-116, August '10.[1]

[1] A portion of this work was done while the first author was employed by Microsoft.

Appendix A: SSH Password Authentication Using Secure Function Evaluation

The following is an unpublished manuscript containing research done under the PIR contract. The authors are Louis Kruger, Matthew Fredrikson, Somesh Jha, and Vitaly Shmatikov, all supported by the contract for the duration of this research.

SSH Password Authentication Using Secure Function Evaluation

Abstract

Over the years, SSH has evolved from a secure alternative to telnet into a robust and extensible protocol that can serve as a secure transport layer for applications that need strong cryptographic security. Interactive password-based authentication, however, remains one of the most popular choices for common SSH deployments, leaving opportunity for an active malicious adversary to either learn the client's password, or impersonate the server or the client. *Password-Authenticated Key Exchange* (PAKE) protocols are designed to be resilient to these attacks. Unfortunately, PAKE protocols are impractical for many common deployment scenarios (*e.g.*, they are not backwards-compatible with legacy password storage schemes, requiring users to re-establish their passwords).

We present a *practical* protocol that provides equivalent guarantees to existing PAKE protocols, but is suitable as a "drop-in" authentication module in existing deployments for SSH, as well as other password-authenticated Internet services. To accomplish this, we use secure function evaluation (SFE) to compare the password credentials between the client and server, embedding computation of the hash function into the comparison protocol. We have implemented an SSH client and server that use our scheme and released an open-source version of the software that is freely available for download.

1 Introduction

Originally designed as a secure alternative to telnet, SSH has since evolved into a layered protocol that serves as the secure transport layer over which many other protocols execute. This functionality has simplified the task of providing cryptographic security to applications that need it. Unfortunately, it has also encouraged SSH deployment in settings where strong authentication mechanisms are not available, or worse yet, take a backseat to more convenient methods such as interactive password login, which is a significant problem. Casual SSH users may assume that the mere presence of SSH guarantees security, unaware of the risks associated with password authentication and improper use of public-key cryptography [37]. It is difficult for a human user to verify authenticity of the server's public key from a hexadecimal fingerprint; in Section 3.1, we describe a man-in-the-middle attack on SSH password authentication which exploits this fact. However, we note that this is not a new attack and was known before. This problem is not limited to SSH, but also affects other Internet services relying on password authentication.

We designed and implemented a practical, yet cryptographically secure protocol for password-based authentication and key establishment in SSH. Even though we use our protocol in the context of SSH, our technique can be applied to any scenario where password-based authentication is necessary. An implementation of our protocol is available at `ANONYMIZED`. Our protocol satisfies three important requirements.

(1) **Compatible with legacy infrastructure.** Our protocol is compatible with existing password authentication infrastructures. It does not require any changes to *legacy servers* beyond upgrading the SSH software and is thus deployable in common settings. The use of cryptographic hash databases to store passwords is common practice on both Unix and Windows systems [34]. Typical Linux systems (current versions of Ubuntu [35], RedHat [33], and Debian [11]) typically use either MD5, or SHA-512 hash function, with salts and iterated rounds for added security against

offline brute-force attacks. Current versions of Windows use a proprietary NT Hash technology [31], but the principle is identical. Our protocol is specifically designed to support storage of passwords in hashed form.

By contrast, other solutions for password-authenticated key exchange require users to re-generate passwords, which greatly limits their deployability. They cannot be installed on legacy servers with large existing user bases. Some also require additional information to be stored on the server or assume the existence of public-key infrastructure (PKI).

(2) Does not decrease security of password storage. At the very least, the password authentication mechanism should not provide weaker security guarantees than the current system, in which users' passwords are stored on the server in hashed form. If the server stores passwords in the clear, a compromise of the server will reveal the passwords of all users. Even without an external attack, a malicious server operator may impersonate a user in other authentication domains.

Our protocol takes as inputs the password from the user and the hashed password from the server (it is essential that the user's input be the actual password and not a hash; otherwise, a malicious server could impersonate the user). Therefore, from the viewpoint of password security, it is as strong as existing solutions, while providing significantly more protection against man-in-the-middle attacks.

(3) Enables derivation of a secure, shared cryptographic key. Our protocol enables the user and the server to derive a shared cryptographic key(s) which can be used to protect their subsequent communications. The key remains secure (*i.e.*, indistinguishable from random) even in the presence of a malicious man-in-the-middle adversary. Unlike existing methods for password authentication in SSH, our protocol does not require the user to check the validity of the server's public key by manually verifying its fingerprint (we argue that this requirement is largely ignored in practical deployment scenarios).

Against an active adversary, the protocol is as secure as can be hoped for in the case of password-based authentication. It does not leak any information except the outcome of an authentication attempt, *i.e.*, for any given password, the adversary can check whether the password is correct. Brute-force password-cracking remains feasible, but every attempt requires executing an instance of the protocol.

Exploiting the special features of password authentication. Our protocol uses Yao's "garbled circuits" protocol for secure function evaluation (SFE) as a basic building block. SFE is used to compute the hash of the SSH client's password and compare it for equality with the hash value provided by the SSH server.

Yao's original protocol is only secure against passive or semi-honest adversaries [27, 39], *i.e.*, if all participants faithfully follow the protocol. This model is clearly unsuitable for SSH, which must be secure even if one of the participants maliciously deviates from the protocol specification. This includes the case when a malicious SSH client—who constructs the garbled circuits in our protocol—deliberately creates a faulty circuit in an attempt to learn the server's input into the protocol. For example, the client may put malformed ciphertexts into the rows of the garbled truth table which will only be evaluated when a certain input bit from the server is equal to "1," and correct ciphertexts into the rows which will be evaluated when this bit is equal to "0." By observing whether the server's evaluation of this circuit fails or not, the malicious client can learn the value of the bit in question. The malicious client may also submit a circuit which computes something other than the hash-and-check-for-equality function required by SSH authentication.

Yao's protocol can be modified to achieve security against malicious participants—either via cut-and-choose techniques [26, 36], or via special-purpose zero-knowledge proofs [22] which enable the server to verify that the circuit is well-formed—but the resulting constructions, while more efficient than generic transformations, are still too expensive for practical use.

Our SFE-based construction in this paper exploits the special structure of the authentication problem in a fundamental way. The purpose of the password authentication subprotocol in SSH is to compute a single bit for the client: whether the hash of the password submitted by the client is equal to the value submitted by the server or not. The standard cut-and-choose construction for SFE in the malicious model requires that the server evaluate several garbled circuits submitted by the client and the majority of them must be correct [26]. In the context of password authentication for SSH, it is sufficient that a *single* circuit be correct. Even if all but one circuits evaluated by the server are faulty, a malicious client does not learn any more than he would have been learned simply by submitting a wrong password.

Our key observation is that to prevent a malicious client from authenticating without the correct password, it is sufficient for the SSH server to either (a) detect that one of the circuits submitted by the client is incorrect, or (b) evaluate at least one correct circuit. In other words, the SSH server either detects the client's misbehavior, or rejects the client's candidate password because its hash does not match the server's value. In either case, authentication attempt is rejected.

We prove the security of our protocol against malicious clients in a (modified) *covert* model of secure computation [5, 20]. Security in the covert model guarantees that any deviation from the protocol will be detected with a high probability. In our proof, instead, we show that, with high probability, either the deviation is detected, or the protocol computes the same value as it would have computed had the client behaved correctly. Security in this model can be achieved at a lower cost than "standard" security against malicious participants, enabling significant performance gains for our implementation viz. off-the-shelf SFE.

Security of an honest client against a malicious SSH server follows directly from the security of the underlying oblivious transfer (OT) protocol against malicious choosers, since the server's input into the protocol is limited to his acting as a chooser in the OT executed as part of Yao's protocol. While the server can always perform a denial-of-service attack by refusing to communicate the result of authentication to the client, this is inevitable in any client-server architecture.

The protocol is secure against replay attacks, since a man-in-the-middle eavesdropper on an instance of the protocol does not learn anything about the client's input (password), server's input (password hash), or the shared key established by the client and the server. Furthermore, we show that even if a man-in-the-middle attacker tampers with the protocol execution, he does not learn more than he would have learned simply by attempting to authenticate with a wrong password.

PAKE protocols. Bellovin and Merritt pioneered a class of protocols that use the client password as a shared secret for mutual authentication [7]. These protocols, commonly referred to as PAKE (*Password-Authenticated Key Exchange*), are resistant to the password compromise scenario described above, even when the client is communicating directly with a malicious impersonator. Furthermore, these protocols alert the client to the presence of an impersonator, allowing the SSH user to cut further communications in high-risk situations. However, existing PAKE protocols are difficult to deploy in many settings, especially when legacy servers and legacy hashed-password files are involved (see Section 2).

In this paper, we present the first password-based authentication and key establishment protocol to satisfy the three design principles listed above. We show that the secure password storage and the secure key establishment requirements can be achieved by comparing the authentication credentials of the user and the server using *secure function evaluation* (SFE) [38], in a legacy-compatible manner.

The main insight that enables backward compatibility with existing infrastructures is that SFE gives the protocol complete flexibility to compute arbitrary hash functions while performing authentication. This makes our protocol suitable as a "drop-in" authentication module in most legacy environments, requiring only that the server and client software be updated to use the new protocol.

Organization of the paper. In Section 2, we discuss related work, and explain why existing PAKE protocols are not suitable for SSH in terms of the three requirements listed in the introduction. In Section 3, we present a technical overview of our problem setting, as well as our proposed solution. In Section 4, we describe the design and implementation of our scheme, and in Section 5 we evaluate it.

	Legacy compatibility	*Secure password storage*	*Mutual authentication*
EKE	*X*	*X*	✓
AEKE	*X*	✓	✓
Ω-Method	*X*	✓	✓
Multiple-Server	*X*	✓	✓

Figure 1: A comparison of existing PAKE protocols. The protocols listed are EKE [7], AEKE [8], Ω-method [17], and Multiple-Server [12]. There are a number of protocols in the literature similar in nature to EKE and AEKE; these are referenced in the text but left out of this table for the sake of clarity.

2 Related Work

Password-Authenticated Key Exchange (PAKE) is a class of password-based authentication protocols designed to be secure even against active adversaries. There are many PAKE protocols in the literature. The first PAKE protocol was described by Bellovin and Merritt as *Encrypted Key Exchange* (EKE) [7]. EKE allows two parties to communicate securely using a weak secret, such as a human-memorable password. The authors observed that a standard symmetric cryptosystem keyed on the weak secret does not provide strong security, and instead proposed to use a temporary asymmetric key pair to exchange a stronger symmetric key for use in the rest of the session (unlike passwords, bitstrings used as keys in common asymmetric schemes are essentially random and are thus difficult to verify by brute-force analysis of protocol messages). EKE provides basic mutual authentication and satisfies our requirement *(3)*. It does not satisfy *(1)* or *(2)*, because in standard deployment for password authentication, the server stores only the hashes of clients' passwords. A number of subsequent protocols have the same properties in terms of our requirements [1–3, 6, 9, 14–16, 24, 25, 28, 42].

Bellovin and Merritt developed *Augmented Encrypted Key Exchange* (AEKE) [8] to enable the server to store only the hash of the password, to protect the latter in case the password database is compromised by an adversary. To prevent impersonation of the client by such an adversary, the protocol uses schemes which allow one party to verify that the other knows both the password and its hash. The first scheme uses a class of *commutative one-way hash functions*. However, there are no known hash functions with the information-hiding properties required to guarantee the security of the protocol, making this scheme of theoretical interest only. The second scheme uses the hashed password stored by the server as the public key in a digital signature scheme. To prove his knowledge of the password, the client signs the session key with the corresponding private key. AEKE implemented with this scheme satisfies our requirements *(2)* and *(3)*. Unfortunately, using passwords stored on the server as signature keys requires substantial changes to the authentication infrastructure and violates requirement *(1)*.

Gentry *et al.* [17] proposed the Ω-method for adding security against server compromise to an arbitrary PAKE protocol. However, all known feasible implementations of their method require the server to store additional information, namely a public/private key pair with the secret key encrypted. Thus, applying this method to one of the previously described protocols will result in a set of implementation constraints basically equivalent to AEKE [8], and ultimately fail to satisfy our requirement *(1)*.

Ford and Kaliski [12] presented a PAKE protocol that protects the secrecy of the client's password against server compromise by distributing it among many servers. When the client authenticates, it interacts with each server to establish a set of strong secrets, after which the servers collaborate to validate the client's identity. A number of subsequent protocols adopt this basic functionality; MacKenzie *et al.* generalize the protocol to a threshold setting [29], and Brainard *et al.* present a lightweight protocol that reduces the computation load on the client [10]. While these protocols satisfy our security requirements (*(2)* and *(3)*), they have the obvious drawback of requiring a specific server-side architecture that may not be common in many settings, and thus fail to satisfy requirement *(1)*.

3 SSH Protocol Overview

The SSH protocol enables secure network services, including remote login and traffic tunneling, over insecure networks such as the Internet [40]. The functionality of the protocol is partitioned into three layers, with each layer defined in terms of messages from the layer beneath it.

- The *transport layer* provides privacy, integrity, and server authentication for the user authentication protocols, as well as the application connection protocols, running on top of it. In short, it provides the layers above it with a plaintext interface for sending encrypted packets reliably over the network.
- The *user authentication layer* authenticates the client to the server. In keeping with the modular design of the protocol, this layer is extensible to a number of authentication mechanisms. The specification for this layer includes public key, password, and host-based authentication subprotocols [41]. However, the majority of deployments use password-based authentication for its convenience and simplicity.
- The *connection layer* multiplexes many distinct communication channels over the SSH transport layer. Several channel types have been defined for various applications, including terminal shell channels for remote login, and traffic forwarding channels for encrypted tunnels.

A typical SSH session proceeds by working through these layers in sequence: first, the SSH transport layer is established, after which the user is able to securely authenticate to the server, and finally the application-specific connections are initiated over the transport layer.

Session Initialization: To establish the SSH transport layer, the server and client must *(1)* perform a key exchange to establish a shared secret that is used to encrypt future communications, and *(2)* validate the server's key, to prevent a man-in-the-middle attack. The SSH specification includes a single Diffie-Hellman group for key exchange [40], although later proposals have extended this layer to allow new Diffie-Hellman groups to be added as needed [13]. As described in Section 1, the host's key is validated by querying the user. Thus, this layer is responsible for the vulnerability described in Section 1.

User Authentication: When the SSH transport layer has been established, the client and server have a secure channel over which they can communicate, and the server has supposedly been authenticated to the client. However, most applications require the client to authenticate to the server. The user authentication layer handles this in an extensible way, by defining a set of messages that can be used to relay general authentication data. The specification describes several mechanisms for authentication, including the username/password method familiar to all users of SSH, as well as public key-based authentication [41]. However, as long as the server and client software can agree on an authentication method, it is straightforward to extend this layer to use new mechanisms that provide better security. For example, Yang and Shieh proposed the use of smart cards for authentication [37], which has subsequently been implemented in at least one SSH software package [32]. Our proposed protocol fits into the SSH protocol in this layer.

When password authentication is used, the protocol proceeds as follows (depicted in Figure 2):

1. The client sends to the server a message of type `SSH_MSG_USERAUTH_REQUEST`, containing the username and password given by the user.
2. Based on the contents of the `SSH_MSG_USERAUTH_REQUEST`, the server responds to the client in one of two ways:
   - If password authentication is disallowed, or the username/password combination supplied is incorrect, then the server responds with an `SSH_MSG_USERAUTH_FAILURE` message.
   - If password authentication is allowed, and the username/password combination is valid, then the server responds with an `SSH_MSG_USERAUTH_SUCCESS` message.

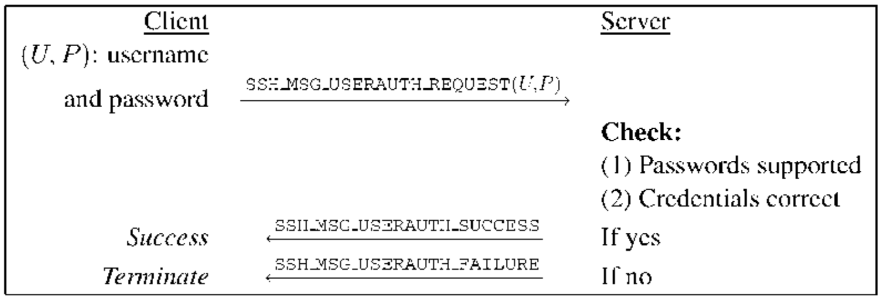

Figure 2: The SSH user authentication sub-protocol.

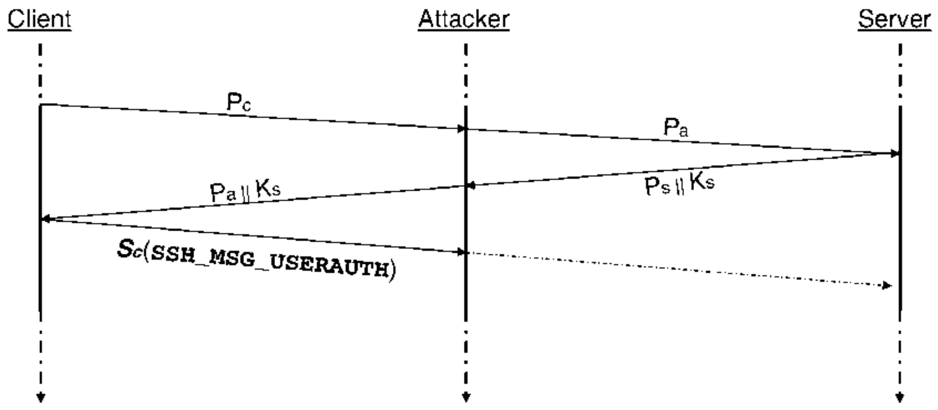

Figure 3: Man-in-the-middle attack on the SSH authentication protocol using Diffie-Hellman key exchange as described in the SSH-2.0 specification. Using this attack, an adversary can learn the client's password, eavesdrop on all communications between client and server, and impersonate the server.

3. If the server sends SSH_MSG_USERAUTH_SUCCESS, then the client has successfully authenticated and may begin requesting services. Otherwise, the protocol terminates.

3.1 Man-in-the-middle attack on conventional SSH password authentication

When SSH password authentication is used, the client and server first negotiate an encrypted tunnel, over which the client sends the password for verification. If an attacker was somehow able to eavesdrop on this encrypted tunnel, the password itself would be revealed to the attacker, who would then be able to impersonate the client at will. A man-in-the-middle attack on the encrypted tunnel, if successful, would allow such a password interception. To prevent such an attack, SSH relies on *host keys* [41] to authenticate the server. However, host keys alone do not entirely solve the problem, as it is necessary to authenticate each server key when a session is initiated. Under certain circumstances, an attacker may still be able to mount a successful attack. Figure 3 depicts this situation when the Diffie-Hellman key exchange is used:

1. After the client and server agree on a key exchange protocol, the client attempts to send the public exponentiated integer P_c to the client.
2. The attacker intercepts P_c, replaces it with a value known to him, P_a, and sends it to the client.

3. The server sends his public integer, along with a host key that is supposed to prove his identity: $P_s \parallel K_s$.

4. The attacker intercepts $P_s \parallel K_s$ and sends $P_a \parallel K_a$ to the server.

5. Using the exchanged public keys, the attacker constructs separate shared secrets S_c and S_s with the client and server, to be used for further communications.

6. The client's software hashes the key that it receives, K_a, and checks a local keystore to see if the hash is recognized. If the client does not have the real servers's public key K_a in his keystore, then the client software asks the user to verify the server key's authenticity:

   ```
   The authenticity of host 'server (1.2.3.4)' can't be
   established. RSA key fingerprint is
   3f:76:22:43:c2:03:b9:71:b0:31:ce:87:37:45:cb:02.
   Are you sure you want to continue connecting (yes/no)?
   ```

   On the other hand, even if the user knows the correct server key, he may assume the key has changed for a non-malicious reason, such as a software upgrade, and allow the connection to proceed.

7. The user validates the authenticity of the key based on its hexadecimal fingerprint, thereby mistakenly asserting that the attacker is the authentic server.

8. The client attempts to send S_c(SSH_MSG_USERAUTH) to the server, containing login credentials encrypted with the shared secret S_c. In this case, the credentials consist of a username and password.

9. The attacker receives S_c(SSH_MSG_USERAUTH), and is able to decrypt it to read the password in clear text.

Critical to the success of this attack is that the user validates the authenticity of the attacker's public key as the server's, in step *(7), an action which we assert is highly probable.* Any OpenSSH user is familiar with the message displayed in step *(6)* – according to the SSH protocol RFC, the fingerprint "...can easily be verified by using telephone or other external communication channels." [40] Not surprisingly, recent research has indicated that one can expect the average user to simply *click through* this dialog without going to such trouble [4], thus accepting the attacker's key; this undermines the very purpose of presenting host key fingerprints to the user. Although the designers of the SSH protocol were aware of this problem when they released the specification, they assumed that widespread future PKI deployment would make it unimportant [40].

4 Protocols

We need a protocol that provides the following functionality: given the client's input x (presumably the password) and the server's input y (presumably hash of the password), the two parties would like to jointly compute whether $H(x) = y$, for some hash function H. In other words, we need to a protocol for the following functionality:

$$(x, y) \longmapsto (\delta(H(x), y), \delta(H(x), y))$$

Where $\delta(a, b)$ is equal to 1 if $a = b$, otherwise it is 0. The first question we must answer is: under which model should our protocol be secure? In the semi-honest model, the adversaries follow the correct protocol, but might try to infer additional information from the messages exchanged during the protocol. The classic protocol presented by Yao [39] can be used to produce a protocol for our problem that is secure in the semi-honest model. An extensive treatment of Yao's protocol along with a proof of correctness is given in [27]. However, the semi-honest model is not suitable in our context, because SSH is frequently used over wide-area networks (WAN) where we cannot expect the parties to obey the semi-honest model.

In the malicious model the adversaries may behave arbitrarily, i.e., lie about their inputs, abort, or not follow the instructions of the protocol. Given a protocol that is secure in the semi-honest model, the protocol can be transformed into a protocol secure in the malicious model [18, 19]. However, the resulting protocols are very inefficient. Lindell and Pinkas [26] present a more efficient protocol that is based on the informal cut-and-choose technique for the two-party case that is secure in the malicious model. However, their protocol is also too slow for our purposes. Protocols that are secure in the semi-honest model are efficient but not secure in our context. On the other hand, protocols that are secure in the malicious model are too inefficient to be useful in our context.

The adversary model we use in this paper is inspired by the *covert model* of Aumann and Lindell [5]. In the covert model, any attempt to cheat by the malicious protocol participant $\mathcal{A}$ is detected by the honest parties with probability at least ϵ. In our model, we demonstrate that if a malicious SSH client cheats, then, with high probability, the SSH server either detects the cheating, or computes exactly the same result it would have computed if the client had not cheated.

4.1 Building Blocks

4.1.1 Oblivious Transfer

In our implementation, we use the oblivious transfer (OT) protocol by Naor-Pinkas [30]. This protocol provides information-theoretic security for the chooser (SSH server in our implementation) and computational security, based on the Diffie-Hellman assumption, for the sender (SSH client). This OT protocol is a good choice for the SSH environment due to its efficiency. As an alternative, we could have implemented our system using a fully simulatable oblivious transfer protocol such as, for example, the new Diffie-Hellman-based OT protocol by Hazay and Lindell [21] whose computational complexity is similar to the Naor-Pinkas protocol. We leave an implementation of this OT protocol as part of our system to future work. The steps of the Naor-Pinkas OT protocol are:

1. Let q be a prime number and let g be a generator for the field $\mathbb{Z}_q$. Elements q and g are known to both parties. The protocol uses a function H which is assumed to be a random oracle. Also, let $s \in \{0, 1\}$ be the chooser's secret choice, and let M_0 and M_1 be the sender's messages. At the end of the protocol, the chooser should only learn M_s and no party should learn any other information.

2. The sender picks a random element $C \in \mathbb{Z}_q$ and sends C to the chooser.

3. The chooser picks a random number $1 \leq k \leq q$ and computes two values: PK_0 and PK_1, where $PK_s = g^k$ and $PK_{1-s} = \frac{C}{g^k}$. The chooser sends PK_0 to the sender.

4. The sender encrypts M_0 and M_1. Specifically, the sender chooses random values r_0 and r_1. It encrypts M_b (where $b \in \{0, 1\}$) as $E_b = \langle g^{r_b}, H(PK_b^{r_b}) \oplus M_b \rangle$ and sends E_0 and E_1 to the chooser.

5. The chooser can compute $H((g^{r_s})^k) = H(PK_s^{r_s})$ and hence decrypt E_s and compute M_s. Intuitively, the chooser cannot decrypt E_{1-s} unless the chooser can find k' such that $g^{k'} = \frac{C}{g^k}$. Therefore, the security of Naor-Pinkas OT depends on the computational Diffie-Hellman assumption.

Note that the Noar-Pinkas OT protocol is not resilient to a man-in-the-middle (MITM) attack, i.e., an attacker can change the second component of E_b so that the chooser receives a message M_b' of attackers choosing. To counter this attack we change E_b to $\langle g^{r_b}, H(PK_b^{r_b}) \oplus (M_b \| H(M_b)) \rangle$

4.1.2 Secure-Function Evaluation (SFE)

Consider any Boolean circuit C, and two parties (Alice and Bob), who wish to evaluate C on their respective inputs x and y. In Yao's "garbled circuits" method [39], Alice transforms the circuit in such a way that Bob can evaluate it obliviously, i.e., without learning Alice's inputs or the values on any internal circuit wire except the output wires. The various steps are as follows:

1. For each wire i of the circuit Alice generates two random keys $k_{i,0}$ and $k_{i,1}$ corresponding to 0 and 1. For all wires in the circuit except the input wires, the truth table for the corresponding Boolean gate is encrypted. If $g(x, y)$ is a gate with input wires j and l, and output wire i, then the truth table value for $g(x, y)$ is encoded as $E_{k_{j,x}}\left(E_{k_{l,y}}\left(k_{i,g(x,y)}\right)\right)$. Here, $k_{j,x}$ is the encryption key for value x of wire j, and similarly for $k_{l,j}$. $k_{i,g(x,y)}$ is the encryption key for the output wire of g with value $g(x, y)$. The four encrypted values representing $g(0, 0)$, $g(0, 1)$, $g(1, 0)$, and $g(1, 1)$ fully specify the gate g. Alice sends the garbled circuit to Bob. Computation of the garbled circuit does not depend on input values and can be performed in advance. However, the same garbled circuit must not be used more than once, or Alice's privacy may be violated.

2. Alice sends the keys corresponding to her own input wires to Bob. Bob obtains the keys corresponding to his input wires from Alice using the oblivious-transfer OT_2^1 protocol. For each of Bob's input wires, Bob acts as the chooser using his corresponding input bit to the function as the choice into OT_2^1, and Alice acts as the sender with the two wire keys for that wire as her inputs into OT_2^1 .

3. Bob evaluates the circuit. Because of the way that the garbled circuit is constructed, Bob, having one wire key for each gate input, can decrypt exactly one row of the garbled truth table and obtain the key encoding the value of the output wire. Yao's protocol maintains the invariant that for every circuit wire, Bob learns exactly one wire key. Because wire keys are random and the mapping from wire keys to values is not known to Bob (except for the wire keys corresponding to his own inputs), this does not leak any information about actual wire values.

After these steps the circuit has been evaluated obliviously by Bob. The final step is for Bob to send to Alice her output wire keys, from which she will learn Alice's designated outputs. A complete description of this protocol along with a formal proof of correctness appears in [27].

A protocol for secure-function evaluation based on Yao's protocol that is secure in the covert model is presented by Aumann and Lindell [5]. Our protocol is similar to the protocol presented by Aumann and Lindell, but uses the context in which the protocol is used.

Our protocol relies on the fact that in the context of password-based authentication, there is no difference from the client's viewpoint between failure due to submitting a wrong password and failure due to a malformed circuit This enables us to achieve security against malicious clients at a much lower cost than the Aumann-Lindell protocol, where the client may learn partial information about the server's input by cleverly creating malformed circuits and seeing which of them were detected. Moreover, by bundling keys in the oblivious-transfer step, any tampering by the man-in-the-middle is detected with high probability.

4.2 Protocol 1: Strawman Protocol

Recall that the client (C) has the password $x = P$ and the server (S) has the hash of the password $y = H(P)$. The protocol works as follows:

- Client hashes the password and obtains $x' = H(x)$.
- Client and server use protocol that is secure in the covert model from [5, Section 6.2] for the function $f(x', y) = \delta(x', y)$.
- After the protocol the client and server know whether their inputs are the same.

The protocol given in [5] has several parameters. Note that the Naor-Pinkas OT protocol provides unconditional security for the server. Therefore, we have the client send the garbled circuits to the server, so the server acts as chooser in the underlying OT-protocol. Moreover, if each bit of the server's input is split into m bits and the cut-and-choose is performed over l circuits, then the protocol is ϵ-deterrent where $\epsilon = (1 - \frac{1}{l})(1 - 2^{-m+1})$.

The straw man protocol has a vulnerability which defeats the entire purpose of storing passwords on the server in the hashed form. To successfully authenticate as a client in this protocol, it is sufficient to know only the hash of the password rather than the password itself. First, this means that if the server is compromised, then the attacker can impersonate any client whose password was stored on the compromised server, even if these passwords were stored in a hashed form. Second, if the server is malicious, then it can impersonate any client who successfully authenticates to it. Nevertheless, the straw man protocol may be useful in certain environments with relaxed security requirements.

4.3 Protocol 2: Main Protocol

We now present our main protocol. Recall that in the SSH context, there are two parties in the protocol: party 1 (client) has input x and party 2 (server) has input y. They want to jointly compute the functionality $(x, y) \longrightarrow (\delta(H(x) = y), \delta(H(x) = y))$ where $\delta_{H(x)=y}$ is equal to 1 if $H(x) = y$; otherwise it is 0. If client gets output of 1, it means that client was authenticated by the server. The reader should interpret x as the password and y as the hash of the password (in other words, the client should only be able to successfully authenticate if he knows the password whose hash matches what the server has). The key idea is that the hash function H is included in the functionality, which makes our protocol resilient against malicious servers impersonating clients (see Section 4.2): knowledge of the password hash is *not* sufficient to authenticate as the client.

The following protocol description assumes that the reader is familiar with the basics of secure-function evaluation (such as garbled circuit construction and oblivious transfer).

- **(Step 1)** Client creates l garbled circuits $C_1, \cdots, C_l$ for $\delta(H(x), y)$. Let server's input $y = y_1 \cdots y_m$ be m bits. The wire keys corresponding to the j-bit of server's input for the i-th garbled circuit C_i is denoted by $k^0_{i,j}$ and $k^1_{i,j}$. Client sends circuits $C_1, C_2, \cdots, C_l$ to the server.

- **(Step 2)** Client and server execute the OT^2_1 protocol m times. In the j-th instance of OT^2_1 the client acts as a sender with inputs $k^0_{1,j} \| k^0_{2,j} \| \cdots \| k^0_{l,j}$ and $k^1_{1,j} \| k^1_{2,j} \| \cdots \| k^1_{l,j}$ and the server acts the chooser with input y_j (the j-th bit of the input). Notice that concatenating the keys prevents the server from learning keys corresponding to different bits, e.g., server cannot learn keys $k^0_{1,j}$ and $k^1_{2,j}$.

- **(Step 3)** Server chooses a random set $S \subseteq \{1, 2, \cdots, l\}$ and sends S to the client.

- **(Step 4)** Client reveals wire keys for circuits C_j such that $j \in S$ to the server (we call this step opening the circuits C_j such that $j \in S$). Client also provides wire keys for its input x for circuits C_j, $j \notin S$.

- **(Step 5)** If the circuits C_j ($j \in S$) are not well-formed (the circuits do not compute $\delta(H(x), y)$ or the keys are not consistent with what was sent in step 2), the server sends 0 to the client. Server computes C_j ($j \notin S$) and obtains answers o_j ($j \notin S$). Server sends $\bigwedge_{j \notin S} o_j$ to the client.

It is clear that if both client and server are honest, then the client will successfully authenticate to the server if and only if it has a password x whose hash $H(x)$ is equal to the input of the server y. Some of the important features of our protocol are:

- The server learns the wire keys corresponding to *one input*. In other words, it is not possible for the server to evaluate circuit C_i on input x_1 and circuit C_j ($j \neq i$) on a different input x_2. This is the rationale behind concatenating the wire keys in step 2 of the protocol. It ensures that a malicious server cannot enter more than one password hash into the computation in an attempt to learn the client's input.

- Assume that out of the l garbled circuits $C_1, \cdots, C_l$ the circuits with index $j \in B$ (where $B \subseteq \{1, 2, \cdots, l\}$) are not valid. The only way the client's cheating is not detected is if B is a subset of $\neg S$ (the complement of S), *i.e.*, all invalid circuits are in the unopened set.

- The server's response to the client is computed as the $\wedge$ of the outputs of all unopened circuits (Step 5). This exploits the essential feature of password authentication, namely, that the client receives a single bit from the server.

  As long as the password submitted by the client is wrong and at least *one* of the unopened circuits is correct (*i.e.*, it correctly computes the hash of the client's input and compares it for equality with the server's input), the server's answer will be 0: "failed authentication attempt." Therefore, a malicious client does not learn anything by submitting invalid circuits, unless *all* unopened circuits are invalid. The outputs of the invalid circuits are effectively hidden from the client by the output of a single correct circuit. By contrast, the generic construction for the malicious model [26] requires that the majority of unopened circuits be correct to prevent information leakages.

  If the client's input is the correct password (*i.e.*, its hash is equal to the server's input), then the client can compute the server's input on his own. Therefore, the client cannot possibly learn anything from the protocol execution, except a single bit confirming that his input is correct.

  Observe that a malicious client who does not know the password will successfully authenticate (*i.e.*, receive bit 1 rather than 0 as his output of the protocol) if and only if *all* unopened circuits are invalid, *i.e.*, the set of invalid circuits B is exactly ¬S. Because S is chosen randomly, the probability of this event is 2^{-l}.

- There is no consistency check on the client's inputs. A malicious client may input different passwords into different circuits. Recall that with high probability, the unopened set contains at least one correct circuit. Clearly, submitting a wrong password to a correct circuit will result in authentication failure. Therefore, the only situation in which the client will authenticate is if he consistently submits the correct password to every correctly formed, unopened circuit. We argue that this is equivalent to knowing the correct password in the first place, *i.e.*, submitting inconsistent inputs does not offer any benefits to a malicious client.

  If the client submits inconsistent inputs and authentication fails, the client does not learn which of the inputs were correct and which were incorrect. Therefore, the client is still limited to a single password per authentication attempt.

We formally argue the protocol preserves the privacy of both parties' inputs.

Client's privacy: Assume that the client is honest and the server is controlled by an adversary $\mathcal{A}$. $\mathcal{A}$'s view consists of the l garbled circuits $C_1, \cdots, C_l$, messages received during the m OT_1^2 protocols, all keys for C_j ($j \in S$), where $S \subseteq \{1, 2, \cdots, l\}$ is chosen by $\mathcal{A}$), and keys corresponding to the client's input x for circuits C_j ($j \notin S$). Assume that views corresponding to the m OT_1^2 protocols only reveal the secrets corresponding to the input y of $\mathcal{A}$ (let the $\mathcal{A}$ input be $y = y_1 \cdots y_m$ then the server learns $k_{j,k}^{y_k}$, $1 \leq j \leq l$ and $1 \leq k \leq m$). This follows from the privacy of the underlying oblivious-transfer protocol. For example, if one uses the Naor-Pinkas oblivious-transfer protocol, then we have the information-theoretic security for the server. Assuming that the encryption scheme used to construct the garbled circuits is semantically secure, revealing the wire keys for circuits C_j ($j \in S$) does not reveal any information about the client's input. Consider the circuits C_j ($j \notin S$). Server can evaluate this circuit on (x, y) but learns nothing else. Consider an ensemble of garbled circuits C'_j ($j \notin S$), where C'_j computes the constant function $\delta(H(x), y)$.[1] If the encryption-scheme is semantically secure, then $\mathcal{A}$

[1]We assume C'_j is constructed from the same encryption scheme that was used to construct $C_1, \cdots, C_j$.

cannot distinguish between circuits C_j and C'_j (for $j \notin S$). Essentially $\mathcal{A}$ only learns whether hash of client's password is equal to its input and nothing else.

Server's privacy: First we give an informal sketch for server's privacy. Assume that server's input is y. We now show that in order for a malicious client who does not know a password x' such that $H(x') = y$, his probability of successful authentication to the server (or impersonation) is no better than $2^{\ l}$. In other words, *the probability that a malicious client who does not know the pre-image of the server's input successfully impersonating a honest client is bounded by* 2^{-l}. The use of even modestly large value of parameter l will make it more likely that the adversary can simply guess the password than to break the protocol. We consider this sufficient, but if desired, extremely large values of l can be used to make the probability of breaking the protocol negligible, with a performance penalty linear in the value of l.

In particular, we show that that the protocol is secure unless the client perfectly guesses the subset S of the l circuits that the server will choose and prepares the encrypted circuits accordingly.

It is sufficient to assume that the malicious client does not know the correct password. If the client knows the password then there is no information to be learned from the server that he does not already possess. There is no useful purpose to cheating the protocol, since he would achieve the desired outcome by executing it faithfully. In this case we simply do not care if the client cheats because he only hurts himself. There are three possible cases:

- **(Case 1)** The client includes an invalid circuit in S. The server will detect this in and reject the authentication in step 5.
- **(Case 2)** Every circuit in S is correct and $\neg S$ (which denotes the complement of S) includes at least one valid circuit. When the server evaluates this circuit, it will evaluate to 0 and the server will reject the authentication. Recall that if a circuit is valid, it will evaluate to 0 on the inputs of the client and server because the client does not know the pre-image of the server's input.
- **(Case 3)** Every circuit in S is correct and every circuit in $\neg S$ is incorrect. We make no claims of correctness about this case. In particular, the client could have made every circuit in $\neg S$ evaluate to 1, in which case the server would accept the impersonating client as authentic.

Since the server chooses the subset S uniformly from the space of proper subsets, the probability of case 3 happening is $2^{\ l}$.

Assume that the server is honest and the client is malicious. The client is controlled by an adversary $\mathcal{A}$. We construct a simulator Sim which works in the ideal model. Recall that in the ideal model the joint computation is performed using a trusted party (TP). Sim acts as the server for $\mathcal{A}$.

- $\mathcal{A}$ sends l copies of the garbled circuits $C_1, \cdots, C_l$ to Sim.
- Sim acts as TP for $\mathcal{A}$ for the m oblivious transfer protocols. Sim knows the inputs of $\mathcal{A}$ (which in the case of the honest client's are the wire keys corresponding to the server's inputs).
- Sim chooses a random set $S_1 \subseteq \{1, 2, \cdots, l\}$ and sends it to $\mathcal{A}$.
- $\mathcal{A}$ sends all the wire keys corresponding to circuits C_j ($j \in S_1$).
- Sim rewinds $\mathcal{A}$, and sends the complement of S_1 to $\mathcal{A}$. $\mathcal{A}$ sends all the wire keys corresponding to circuits C_j ($j \notin S_1$). Note that after this step Sim knows the wire keys corresponding to all the garbled circuits $C_1, \cdots, C_l$.
- Sim rewinds $\mathcal{A}$, picks a random set $S \subseteq \{1, 2, \cdots, l\}$, and sends it to $\mathcal{A}$. $\mathcal{A}$ sends wire keys corresponding to the circuits C_j ($j \in S$).
- $\mathcal{A}$ provides the wire keys for all circuits C_j ($j \notin S$). Note that since Sim knows the wire keys for all the garbled circuits, it can now construct $\mathcal{A}$'s inputs x_j to circuits C_j ($j \notin S$). If the inputs are

inconsistent (*i.e.*, not all equal to the same value), Sim sends 0 to $\mathcal{A}$. If all inputs x_j ($j \notin S$) are equal to x, then Sim sends x to the TP. If TP returns 1 (which means that $\mathcal{A}$ knew the pre-image of server's input), then Sim sends 1 to $\mathcal{A}$ (which essentially means that the malicious client was authenticated). If TP returns 0, we proceed to the next step.

- Sim checks the validity of all the circuits C_j ($j \in S$). If any of these circuits is found to be invalid, then Sim sends 0 to $\mathcal{A}$. Otherwise Sim sends 1 to $\mathcal{A}$.

Assume that $\mathcal{A}$ does not know the pre-image corresponding to the server's input. Suppose the inputs x_j ($j \notin S$) are not equal. In this case, $\mathcal{A}$ receives 0 from Sim. We argue that in the real model $\mathcal{A}$ would receive 1 only if it knows the pre-image corresponding to the server's input (a contradiction). The only way $\mathcal{A}$ receives a 1 if all of his inputs into correctly formed, unopened circuits are pre-images of the server's input, which means $\mathcal{A}$ knew the pre-image of the server's input to begin with, contradicting our assumption. Hence the views of $\mathcal{A}$ in the ideal and real world are the same when the inputs x_j ($j \notin S$) are not equal, unless all opened circuits are valid *and* all of the unopened circuits are invalid (the probability of this event is 2^{-l}).

Now assume that inputs x_j ($j \notin S$) are all equal to x. Let E_1 be the event that a honest server denies authentication to $\mathcal{A}$ in the real model, and E_2 be the event that Sim denies authentication to $\mathcal{A}$ in the ideal model. Recall that denying authentication is tantamount to $\mathcal{A}$ receiving 0. It is easy to see that if E_1 and E_2 are true, then the view of $\mathcal{A}$ in the real model is indistinguishable from the view of $\mathcal{A}$ in the ideal model. The probability of event $E_1 \wedge E_2$ not happening is bounded by 2^{-l-1}.

We conclude that if the client does not know the pre-image of the server's input, then the probability that the view of $\mathcal{A}$ in the real model is indistinguishable from the view of $\mathcal{A}$ in the ideal model with probability atleast $1 - 2^{-l+1}$. In other words, conditioned on the event that the malicious client does not know the pre-image of the server's input, the probability of the simulator failing is bounded by 2^{-l+1}. This model is very similar to the "failed simulation" model given by Aumann and Lindell [5, Section 3.2].

4.3.1 Security against man-in-the-middle attacks

Observing an instance of our protocol yields no information that will be useful in subsequent instances of the protocol (*e.g.*, it does not reveal the parties' inputs). Consider a man-in-the-middle (MITM) attacker who captures all messages exchanged between the client and the server. The attacker will not be able to replay a message from an old session because various steps in the protocol use fresh, randomly generated values. For example, in each instance of the Naor-Pinkas oblivious-transfer protocol, the chooser (in our case the SSH server) generates a random value k and sends g^k or $\frac{C}{g^k}$ (where g is generator of the underlying group and C is an element in the group). Therefore, observing the client's and server's inputs into our protocol does not reveal the password hash, nor any other information that can be used in subsequent sessions.

Now consider a MITM attacker who attempts to learn information about the inputs by tampering with an instance of the protocol between a valid client (who knows the correct password) and a valid server (who knows the correct password hash). Specifically, MITM attempts to make the authentication attempt fail *conditionally*, depending on a bit of the server's input. If successful, this would leak 1 bit of the password hash, violating security of the protocol. We show that *any* tampering causes the server to reject authentication with near certainty, and is thus equivalent to simply attempting to authenticate with a wrong password.

Note that because the client's inputs into the oblivious transfer protocol are information-theoretically secure, only the server's inputs (*i.e.*, password hash) can be attacked in this way. Without loss of generality, suppose MITM is attempting to learn the first bit of the server's input.

The intuition why this attack doesn't work is as follows. First, any tampering must modify both the circuit(s) and the corresponding wire keys, or it will be detected when the server verifies opened circuits. Second, in the oblivious transfers used by our protocol for each bit of the server input, the wire keys representing this bit in all circuits are "bundled" together and transferred concatenated *as a single value*, with an integrity check. As we show, MITM cannot tamper with or replace the keys used in a single circuit; any tampering will affect *all* circuits received by the server, ensuring detection with a very high probability.

Attack 1. MITM tries to replace circuit C_1 with circuit C_1' where C_1' simply returns the first bit of the password hash. This attack does not work because MITM does not know the wire keys used to construct circuit C_1. There are two cases:

1. $C_1 \in S$, the set of opened circuits. In this case, the server will see that C_1' is not a valid circuit and will reject the authentication attempt.

2. $C_1 \notin S$. In this case, when the server attempts to evaluate circuit C_1, the Yao circuit evaluation will fail because the wire keys are invalid. Again, the server will reject the authentication attempt.

Attack 2. MITM tries to tamper with OT_2^1 as a black box. First, MITM impersonates the server in OT_2^1 with the client for the first bit of inputs into each circuit C_i, $1 \leq i \leq l$. This enables MITM to retrieve the wire keys $k_{0,0}^1 \parallel \cdots \parallel k_{0,0}^l$ encoding this bit in all circuits.

MITM lets the OTs between the client and the server proceed normally for the remaining bits $k_1, \cdots k_{b-1}$. For the first bit k_0, MITM impersonates as the sender and lets the server choose between $k_{0,0}^1 \parallel \cdots \parallel k_{0,0}^l$ and $k_{0,1}^{1'} \parallel \cdots \parallel k_{0,1}^{l'}$, where $k_{0,1}^{1'} \parallel \cdots \parallel k_{0,1}^{l'}$ have been generated by MITM.

This attack will fail when the server verifies the circuits in S. Because $k_{0,1}^{1'} \parallel \cdots \parallel k_{0,1}^{l'}$ are not valid wire keys, the server will detect that the circuits in S are malformed and will reject the authentication attempt.

Attack 3. MITM tries to tamper with OT_2^1 by modifying specific messages in the Naor-Pinkas OT protocol.

1. MITM replaces the entire message $k_{0,0}^1 \parallel \cdots \parallel k_{0,0}^l$. This will be detected for the same reason as in Attack 2.

2. MITM replaces some, but not all, of the wire keys $K' \subset \{k_{0,0}^1, \cdots, k_{0,0}^l\}$. Assume MITM attempts to tamper with the 0-wire key only for the first circuit, $k_{0,0}^1$, while leaving the other wire keys intact. Observe, however, that the concatenation $k_{0,0}^1 \parallel \cdots \parallel k_{0,0}^l$ is transferred as a single encrypted value, and in the Naor-Pinkas oblivious transfer, MITM cannot tamper with the transferred value (see Section 4.1.1). Therefore, MITM cannot selectively replace only some of the wire keys: the integrity hash will not match and tampering will be detected by the server, who will always reject the authentication attempt even if the client's original password was correct (thus leaking no information to MITM).

4.4 Protocol 3: Adding key establishment

In the context of SSH, client and server need to compare their inputs and also establish a session key if the comparison between their inputs is successful. The protocol is an easy extension of our main protocol.

- Client picks a random key K. Client and server execute a variation of the main protocol that computes the following functionality:

$$((x, K), y) \rightarrow (\textbf{if } (H(x) = y) \textbf{ then } (K, K) \textbf{ else } (\perp, \perp))$$

A malicious client may pick key K that is not truly random. However, if the client is malicious, it can unencrypt and forward its messages to an adversary regardless of this. On the other hand, suppose the server is malicious. If the malicious server knows the hash $H(x)$ of the client's input x, then it knows

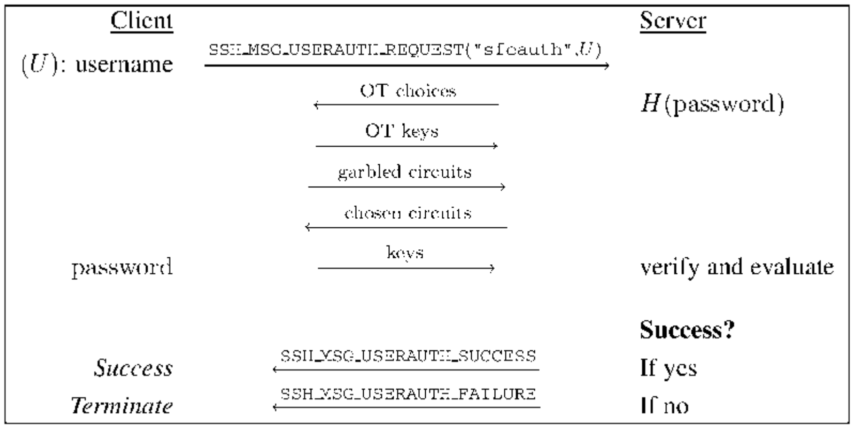

Figure 4: The SFE user authentication protocol 2.

the session key K. In this case, the server can again forward the unencrypted messages to the adversary. If the server does not know $H(x)$, then it cannot obtain the session key. In other words, adding the extra functionality of distributing session keys does not affect the security of protocol 2 (which only compares $H(x)$ and y).

5 Evaluation

We modified an existing open-source SSH client and server to use each of the protocols described in Section 4, and took several performance measurements to evaluate the feasibility of our approach. We implemented and tested the salted MD5 and SHA-512 hashing methods commonly used in Linux distributions. Our findings can be summarized as follows:

- Protocols that are secure in the semi-honest model, which is only secure against passive adversaries, can be executed very quickly.
- Making the protocols secure against active adversaries increases authentication time substantially, depending on the size of the authentication circuit and the security parameter. For example, calculating an MD5 hash using 90 circuits increases the authentication time from around 2 seconds to 12 seconds. Although this may seem tedious for some users, we achieve a high level of security with only modest delays on inexpensive modern hardware. We conclude that the technique can achieve a favorable balance between efficient practicality and high security on systems using MD5 hashing.
- Due the simplicity of private equality testing, Protocol 1 (the Straw man protocol) can run extremely quickly even under the covert model at the expense of resisting impersonation by an adversary who has gained knowledge of a user's password hash. If this security requirement can be relaxed (for example, in an environment where passwords are stored on the server using an identity hash, i.e. in plaintext) protocol 1 could be a useful high-speed authentication protocol.

5.1 Implementation

We implemented the protocols by modifying the Dropbear 0.52 SSH client and server to support a new authentication protocol, to which we assigned the name "sfeauth" in the SSH authentication protocol namespace. The scheme used for incorporating the protocols into the SSH protocol is shown in Figure 4. Our protocol is executed through the encrypted SSH tunnel using a reserved message which we dubbed SSH_MSG_USERAUTH_SFEMSG. If at any time the server detects a cheating attempt by the client, the server denies the authentication and terminates the protocol.

The MD5, SHA-512, and private equality circuits were implemented using a prototype circuit compiler we developed first described in [23], which also contains an embeddable implementation of the Yao "garbled circuit" protocol. The protocol was extended using the techniques due to Lindell and Pinkas [26] to add resistance to malicious parties in the covert model. The implementation also uses the oblivious transfer protocol due to Naor and Pinkas [30]. All of the Yao protocol and authentication code was written in C++ and integrated with the Dropbear SSH client and server.

5.1.1 Optimizations

To improve performance of the protocols, we introduced several optimizations to our implementation.

1. The client computes the garbled circuits used in the authentication protocol in advance of interacting with the online protocol. By precomputing and storing garbled circuits, the time spent in this CPU intensive step is removed from the user's perceived wait time to login to a server.

2. We implemented an optimization described by Goyal *et al.* [20]. In constructing the garbled Yao circuits, the client generates a set of seeds for a cryptographically-secure pseudorandom number generator (PRNG). The circuits are garbled using this PRNG, with one seed per circuit, and hashes of the circuits are sent in place of the whole circuits. After the server has chosen which circuits to open, the seeds for the non-chosen circuits are revealed to the server, who then uses the PRNG to reconstruct the garbled circuit and verify the hash values, and only the circuits to be evaluated are transferred in full. This saves many megabytes of wire communication, improving the overall protocol performance.

The prototype SSH client and server, as well as further documentation, can be downloaded from our project website.[2]

5.2 Experiments

We conducted several usage experiments to measure the performance of the authentication, and determine its feasibility in real settings. Note that the semi-honest version is not secure for real-world usage where the possibility of active malicious adversaries cannot be ruled out, but the experiment is useful to establish an upper bound for the potential performance with further optimizations. The tests were performed over a local network using computers with eight core Intel Xeon processors and 8GB of RAM.

The performance results of our experiments are shown in Table 1. The first row corresponds to the semi-honest version of the protocol, and is the time on which the *ratio to semi-honest* column for other rows is based. The column titled *probability of attack success* refers to the probability of a malicious client successfully convincing the server that he has the proper credentials to authenticate. This calculation is discussed in detail in section 4.3. As our results indicate, the time required to complete the protocol increases linearly as the number of circuits increases, while the security guarantee increases exponentially in this measure. Note that for less than an order of magnitude increase over semi-honest implementation, sufficient security guarantees for many practical settings can be attained, especially when using the MD5 protocol.

The experiments on 120 and 150 circuits for SHA-512 could not be completed. This is due to the complexity of the SHA-512 circuit, which has close to 124,000 gates, compared to under 18,000 for the MD5 circuit. Because of this, our test machines did not have enough RAM to hold 120 or more encrypted copies of the circuit. SHA-512 uses 80 rounds of its compression functions. As a performance optimization, it would be possible to use a reduced-round variant of the circuit into which the client inputs into the circuit the output after the first N rounds, and the circuit computes the remaining $80 - N$ rounds. Such an optimization would reduce the margin of safety built into the SHA-512 hash function against pre-image attacks, if the password database were compromised. However, such an optimization

[2]ANONYMIZED

# Circuits	*MD5 Online Time (sec)*	*MD5 Ratio to Semi-Honest*	*SHA-512 Online Time (sec)*	*SHA-512 Ratio to Semi-Honest*	*Probability of attack success*
1 (*Semi-Honest*)	0.52	1.0	3.02	1.0	100%
30	3.99	7.7	28.8	9.5	$1.86 \times 10^{-7}\%$
60	7.86	15.2	62.6	20.7	$1.73 \times 10^{-16}\%$
90	12.35	23.8	85.0	28.1	$1.62 \times 10^{-25}\%$
120	16.46	31.8	N/A	N/A	$1.50 \times 10^{-34}\%$
150	20.57	39.6	N/A	N/A	$1.40 \times 10^{-43}\%$

Table 1: Wall-clock performance and security guarantees for the optimized protocol in both semi-honest and covert settings. All times are given in seconds.

would reduce the size of the circuit by a factor of $\frac{80-N}{80}$, with a corresponding increase in performance. Naturally, this optimization would need to be weighed carefully against the cryptographic consequences.

Overall, we believe these results indicate that our technique is suitable for common use in real applications, especially on systems based on a simpler hash function than SHA-512, such as the common MD5 standard.

We note that the performance is sensitive to available processing power due to the many cryptographic primitives employed. For example, our implementation takes advantage of parallelization on multi-core processors to encrypt multiple circuits in parallel as the server performs its circuit verifications. Due to the independence of the verification of each circuit, parallel scaling can be achieved that is extremely efficient with respect to available processors, potentially allowing high security authentication with minimal delays to clients on server machines with enough processing power.

Overall, we believe that the results we have achieved so far demonstrate the potential of this technique as a practical and secure addition to the body of research in secure password authentication.

6 Conclusion

In this paper, we have addressed the problem of providing secure mutual authentication for the SSH protocol, maintaining full backwards compatibility with existing server/client infrastructures as a primary requirement. Leveraging the unique flexibility of SFE to compute arbitrary hash functions within the protocol, we constructed a protocol that provides an identical set of security guarantees as previous work in the area, while remaining suitable as a "drop-in" authentication module on nearly all existing servers. Furthermore, we demonstrated that a conservative set of optimizations to the protocol made it practical for common use from a performance standpoint. In the future, we intend to study further optimizations to certain parts of the protocol that improve performance without sacrificing security, as well as additional applications that could benefit from our basic technique.

References

[1] M. Abdalla, E. Bresson, O. Chevassut, B. Möller, and D. Pointcheval. Provably secure password-based authentication in TLS. In *ASIACCS '06: Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pages 35–45, New York, NY, USA, 2006. ACM.

[2] M. Abdalla, D. Catalano, C. Chevalier, and D. Pointcheval. Efficient two-party password-based key exchange protocols in the uc framework. In *CT-RSA*, pages 335–351, 2008.

[3] M. Abdalla and D. Pointchavel. Simple password-based encrypted key exchange protocols. In *CT-RSA 2005*, pages 191–208. Springer, 2005.

[4] A. Adams and M. A. Sasse. Users are not the enemy. *Commun. ACM*, 42(12):40–46, 1999.

[5] Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In *TCC*, pages 137–156, 2007.

[6] M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In *EUROCRYPT*, pages 139–155, 2000.

[7] S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secureagainst dictionary attacks. In *SP '92: Proceedings of the 1992 IEEE Symposium on Security and Privacy*, page 72, Washington, DC, USA, 1992. IEEE Computer Society.

[8] S. M. Bellovin and M. Merritt. Augmented encrypted key exchange: a password-based protocol secure against dictionary attacks and password file compromise. In *CCS '93: Proceedings of the 1st ACM conference on Computer and communications security*, pages 244–250, New York, NY, USA, 1993. ACM.

[9] V. Boyko, P. D. MacKenzie, and S. Patel. Provably secure password-authenticated key exchange using Diffie-Hellman. In *EUROCRYPT*, pages 156–171, 2000.

[10] J. Brainard, A. Juels, B. Kaliski, and M. Szydlo. A new two-server approach for authentication with short secrets. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 14–14, Berkeley, CA, USA, 2003. USENIX Association.

[11] Debian reference manual chapter 4 – authentication. `http://www.debian.org/doc/manuals/debian-reference/ch04.en.html`.

[12] W. Ford and J. Kaliski, B.S. Server-assisted generation of a strong secret from a password. In *IEEE 9th Internation Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 176–180, 2000.

[13] M. Friedl, N. Provos, and W. Simpson. RFC 4419: Diffie-Hellman group exchange for the secure shell (SSH) transport layer protocol (proposed standard), Mar. 2006.

[14] R. Gennaro. Faster and shorter password-authenticated key exchange. In *TCC*, pages 589–606, 2008.

[15] R. Gennaro and Y. Lindell. A framework for password-based authenticated key exchange. In *EUROCRYPT*, pages 524–543, 2003.

[16] C. Gentry, P. D. MacKenzie, and Z. Ramzan. Password authenticated key exchange using hidden smooth subgroups. In *ACM Conference on Computer and Communications Security*, pages 299–309, 2005.

[17] C. Gentry, P. D. MacKenzie, and Z. Ramzan. A method for making password-based key exchange resilient to server compromise. In *CRYPTO: 26th Annual International Cryptology Conference*, pages 142–159, 2006.

[18] O. Goldreich. *Foundations of Cryptography: Volume 2 – Basic applications*. Cambridge University Press, 2004.

[19] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game – a completeness theorem for protocols with honest majority. In *19th STOC*, 1987.

[20] V. Goyal, P. Mohassel, and A. Smith. Efficient two party and multi party computation against covert adversaries. In *EUROCRYPT*, pages 289–306, 2008.

[21] C. Hazay and Y. Lindell. Efficient oblivious transfer with simulation-based security. 2009.

[22] S. Jarecki and V. Shmatikov. Efficient two-party secure computation on committed inputs. In *EUROCRYPT*, pages 97–114, 2007.

[23] S. Jha, L. Kruger, and V. Shmatikov. Towards practical privacy for genomic computation. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 216–230, Washington, DC, USA, 2008. IEEE Computer Society.

[24] J. Katz, R. Ostrovsky, and M. Yung. Efficient password-authenticated key exchange using human-memorable passwords. In *EUROCRYPT '01: Proceedings of the International Conference on the Theory and Application of Cryptographic Techniques*, pages 475–494, London, UK, 2001. Springer-Verlag.

[25] J. Katz, R. Ostrovsky, and M. Yung. Forward secrecy in password-only key exchange protocols. In *SCN*, pages 29–44, 2002.

[26] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT*, pages 52–78, 2007.

[27] Y. Lindell and B. Pinkas. A proof of security of yao's protocol for two-party computation. *Journal of Cryptology*, 22(2), 2009.

[28] P. D. MacKenzie, S. Patel, and R. Swaminathan. Password-authenticated key exchange based on RSA. In *ASIACRYPT*, pages 599–613, 2000.

[29] P. D. MacKenzie, T. Shrimpton, and M. Jakobsson. Threshold password-authenticated key exchange. In *CRYPTO: 22nd Annual International Cryptology Conference*, pages 385–400, 2002.

[30] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *SIAM Symposium on Discrete Algorithms (SODA)*, 2001.

[31] MSDN security account manager protocol specification. `http://msdn.microsoft.com/en-us/library/cc245506(PROT.13).aspx`.

[32] The OpenSC project. `http://www.opensc-project.org/`.

[33] RedHat enterprise linux 5.4 deployment guide – shadow passwords. `http://www.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/5.4/html/Deployment_Guide/s1-users-groups-shadow-utilities.html`.

[34] S. Smith and J. Marchenisi. *The Craft of System Security*. Addison Wesley, 2008.

[35] Ubuntu manual entry on shadow passwords. `http://www.opensc-project.org`.

[36] D. P. Woodruff. Revisiting the efficiency of malicious two-party computation. In *EUROCRYPT*, pages 79–96, 2007.

[37] W.-H. Yang and S.-P. Shieh. Password authentication schemes with smart cards. *Computers & Security*, 18(8):727–733, 1999.

[38] A. C. Yao. Protocols for secure computations. In *FOCS*, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

[39] A. C.-C. Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167, 1986.

[40] T. Ylonen and C. Lonvick. RFC 4251: The secure shell (SSH) protocol architecture, Jan. 2006.

[41] T. Ylonen and C. Lonvick. RFC 4252: The secure shell SSH authentication protocol, Jan. 2006.

[42] M. Zhang. New approaches to password authenticated key exchange based on rsa. In *ASIACRYPT*, pages 230–244, 2004.